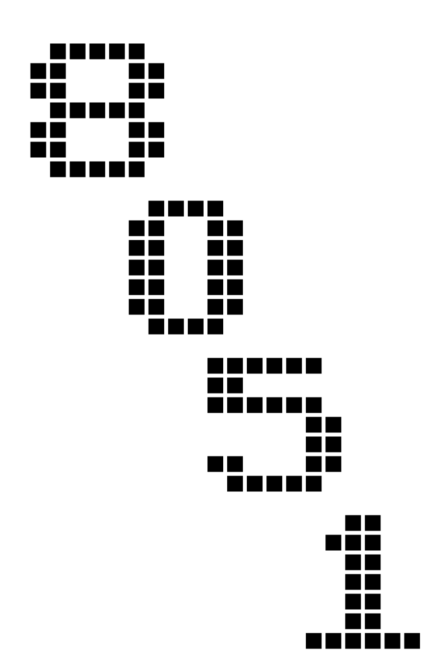

User's
Guide


8051




HI-TECH
SOFTWARE



HI-TECH C for 8051

HI-TECH Software.

Produced on: 20th May 2006

HI-TECH Software Pty. Ltd.
ACN 002 724 549
PO Box 103
Alderley QLD 4051
Australia

email: hitech@.htsoft.com
web: http://www.htsoft.com
ftp: ftp://www.htsoft.com

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Typographic conventions

Different fonts and styles are used throughout this manual to indicate special words or text. Computer prompts, responses and filenames will be printed in `constant-spaced` type. When the filename is the name of a standard header file, the name will be enclosed in angle brackets, e.g. `<stdio.h>`. These header files can be found in the `INCLUDE` directory of your distribution.

Samples of code, C keywords or types, assembler instructions and labels will also be printed in a `constant-space` type. Assembler code is printed in a font similar to that used by C code.

Particularly useful points and new terms will be emphasized using *italicized type*. When part of a term requires substitution, that part should be printed in the appropriate font, but in *italics*. For example: `#include <`*`filename.h`*`>`.

Chapter 2

HI-TIDE Overview

This chapter details the different regions displayed in HI-TIDE's main window as well as general preferences relating to HI-TIDE and its operation.

2.1 Layout Overview

There are several different regions in the HI-TIDE main window. Apart from the window decorations supplied by *Windows* or the window manager you are running, the regions include the *menus*, the *toolbar*, and the *status bar*. These are discussed in Sections 3.1, 3.2 and 3.3, respectively.

Figure 2.1 shows the appearance of HI-TIDE under *Windows* with no project loaded. The menus and toolbar are at the top of the window and the status bar is located along the lower edge of the window. The remainder of the HI-TIDE window is left blank. Most of the functionality of these components is disabled until a project is loaded.

Once a project has been loaded (see Section 3.1.4) the toolbar and status bar activate and HI-TIDE's window is populated with graphical regions, called *areas*, whose state has been read in from the project file. Areas are described in Section 2.2. Figure 2.2 shows how HI-TIDE may typically appear after a project is loaded.

2.2 HI-TIDE Areas

An *area* is a graphical region of the main HI-TIDE window. There are three HI-TIDE areas which can be displayed: the *Project area*, the *Build area* and the *Workspace area*. These areas are visible in Figure 2.2 and have been individually highlighted in Figures 4.1, 4.3 and 2.3.

Figure 2.1: HI-TIDE without a project loaded

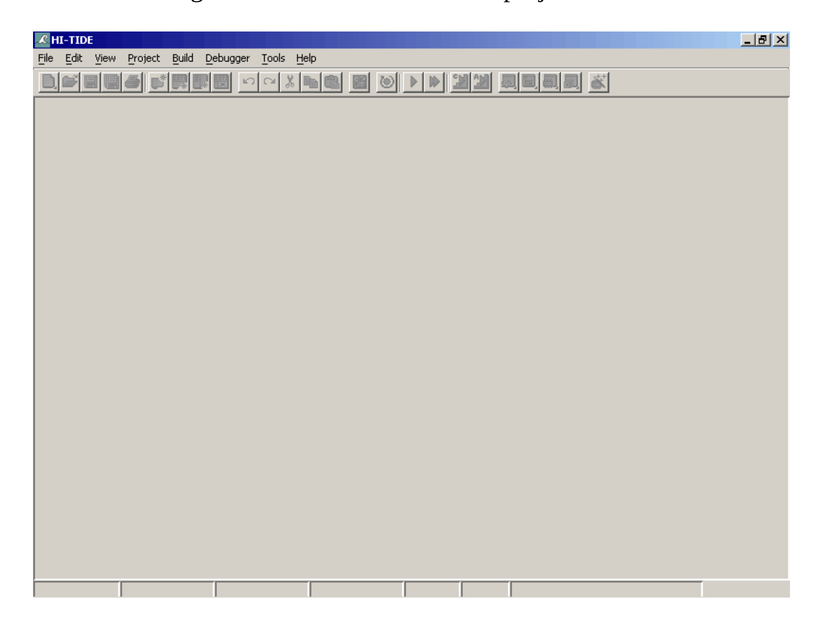


Figure 2.2: Layout overview with project loaded

These areas consume the remainder of HI-TIDE's main window. Borders separate the areas and can be dragged to resize the area's display. As the mouse is moved over the draggable area border, it changes shape to a double-ended arrow. For example, the Project area can be resized by clicking and dragging the divider that separates it from the Workspace area. Changing the size of the Project area will also change the width of the Workspace area.

All areas use a tab viewer to be able to display and manage different contents and its operation is similar to tabbed viewers in other applications. The tabs are located along the bottom of the area. Clicking a tab replaces what is displayed in the area with the new tab pane.

Each tab pane contains one or more *views*. A view is further subdivision of a pane. In some cases a tab may show an *empty view*. This is a view with no contents and it appears as a blank, grey area.

2.2.1 The Project and Build Areas

The Project area and Build area have similar operation. They are static areas, in that their tabbed panes and views are managed by HI-TIDE and cannot be changed by the user.

These areas can be hidden from display to allow the other area(s) to expand. The size of the remaining area(s) automatically adjust as an area is hidden or re-shown. For example, to hide the Project area either click on the close button in the top right corner of the area, or choose **Hide Project View** from the **View** menu. To display the project view select **Show Project View** from the **View** menu. The Build area can be hidden and shown again in a similar way.

2.2.2 The Workspace Area

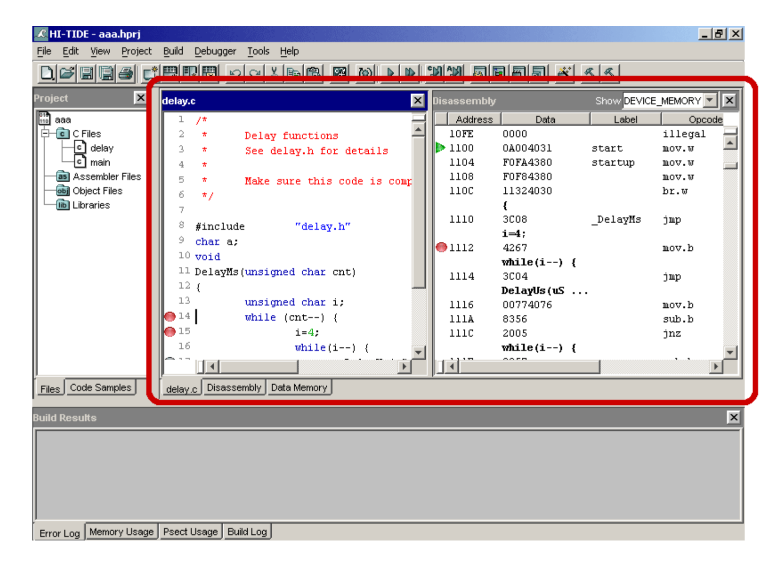
The *Workspace area* (often referred to just the *workspace*) is the main area in which program development and debugging takes place. This is the area where the editor and memory displays etc. are shown. The Workspace area is shown circled in Figure 2.3. Its operation is similar to the other areas, but with some notable differences.

The Workspace area cannot be hidden. There is no close button, nor menu items, which will completely remove it from display.

The Workspace area has tabs for showing multiple panes like the other areas, but these tabs can be managed by the user. The user may add or remove workspace tabs, and tabs may be renamed so that their contents are easily identified. These operations are further described in the following sections.

Figure 2.4 shows a typical Workspace tab, labelled **delay.c**, with an Editor view showing on the left side and an assembly view showing on the right side.

Figure 2.3: Workspace area



2.2.2.1 Adding a Workspace Tab

Views which can be displayed in the Workspace area will generally create and add a tabbed pane themselves. This is described in more detail in Section 2.2.2.1.

Another way to add a tabbed workspace is to select the **New Tab** menu item in the **Views** menu or the **New Tab** toolbar button from the standard toolbar.

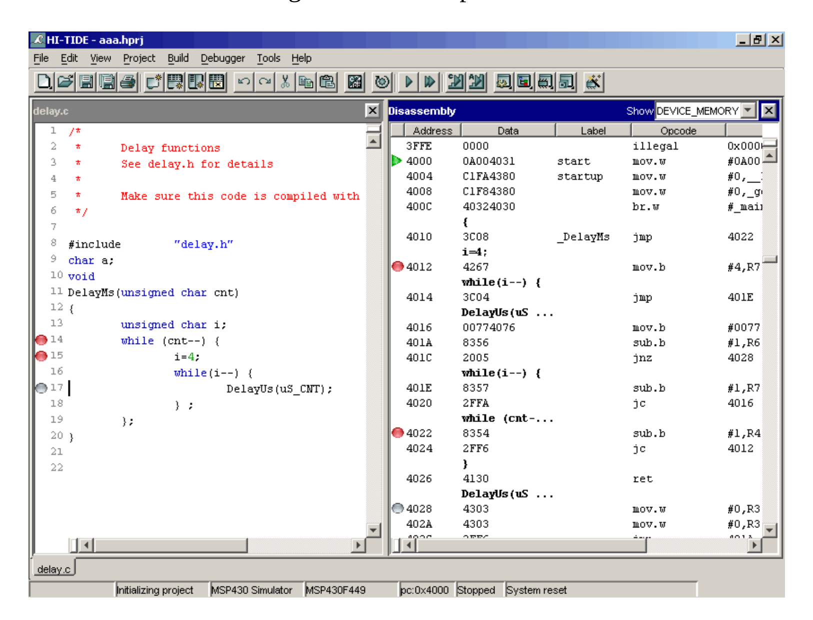
A new tab can also be created and added in by right-clicking on the tabs and selecting **New Tab** from the popup menu.

2.2.2.2 Removing a Workspace Tab

The Workspace tabs can be removed in a number of ways. The first way is to close the view using the **Close View** button located in the top right corner of each view, the **Close View** menu item from the **Views** menu or selecting the **Close View** button from the standard toolbar. If this is the only view in the tab, then the tab will be closed along with the view.

The tabbed workspace can also be closed via the tab workspace popup menu. To close the tab, right-click over the tab and select **Close Tab** from the popup menu. When a tab is closed in this fashion, all the views it contains will be closed as well.

Figure 2.4: Workspace tab



2.2.2.3 Renaming Workspace Tabs

Generally, when a new tab is created by a view, it is labelled with the name of that view. For example, if the tab was created for a file, it will be labelled with the name of the file. Or if it was created for the assembly view, it will be labelled Assembly view. However, if the tab is created from selecting the **New Tab** menu item (from either the **Views** menu or a Workspace view's popup menu), the tab will be labelled **New Tab**.

At times, these tab labels may need to be renamed to better identify the view. This might be the case if the tab workspace contains more than one view.

The tab can be renamed in one of two ways. The first is to right-click on the tab that is to be renamed and select **Rename Tab**. The label of the tab will turn into a text box where the name of the tab can be changed. Pressing *Enter* will apply the changes. Pressing the *Escape* key or clicking outside the text box will cancel the action.

Alternatively, to change the name of the tab, double-click on the label of the tab. The label of the tab will turn into a text box to allow editing of the name of the tab. Pressing *Enter* will apply the changes. Pressing *Escape* or clicking outside the text box will cancel the action.

2.2.3 Workspace Views

Workspace views are the views that can be displayed inside the tabbed Workspace area. They mostly share common properties and functionalities. The following section detail how the Workspace views behave and how they can be adjusted to suite the requirements of the project.

2.2.3.1 Displaying a View

There are several ways in which a Workspace view may be displayed:

- Selecting the view's namesake from the **view** menu.
- Clicking the view's toolbar icon.
- Dragging the view's toolbar icon to the Workspace area.

These methods apply to all Workspace views, except the Editor view which is discussed separately below.

When selecting from the **View** menu or clicking the toolbar icon, the new view will be assigned a new tab in the Workspace area and the contents of that view displayed in this tab. The tab's name will set to the name of the view when the tab is created, but can be changed as described in Section 2.2.2.3.

Dragging the toolbar icon allows for more specific configuration of the Workspace area. A toolbar view icon dragged to an existing view will replace that view with the new contents. Thus,

if a existing view is split (see Section 2.2.3.3) and a view toolbar icon dragged onto the resulting empty view, more than one view can be displayed per Workspace tab.

If a toolbar icon is dragged over an region that cannot display the view, a “no drop” circular symbol with a line through it, will be shown for the mouse pointer while it is over that region. If the mouse is over an region which can accept the view, the mouse pointer will become the file transfer icon. This icon is platform dependent.

Displaying the Editor View The Editor view has neither an entry in the **View** menu, nor a toolbar icon. A new Editor view may be displayed using one of several methods:

- Double-clicking a source file icon in the **Files** or **Code samples** view in the Project area.
- Selecting **Open** from the **File** menu and choosing a file.
- Clicking the **File Open** button from toolbar button and choosing a file.
- Dragging a source file icon from the **Files** or **Code samples** view in the Project area.

In all methods, except when dragging a source file, a new Workspace tab is created for the view, however if there is an existing Editor view of the same file this view will be displayed rather than a new view. This applies regardless of whether the existing Editor view is in a Workspace tab by itself, or is one of several views within a Workspace tab. If more than one Editor view of the same file exists and the currently selected tab doesn’t contain one of these views, the first tab (from left to right) in the Workspace area with the Editor view is selected and shown.

In a similar way to other views, if a source file icon is dragged to an existing view that view will be replaced with the new contents. If a existing view is split and a source file icon icon is dragged onto the resulting empty view, an Editor view can be shown with other views within a Workspace tab.

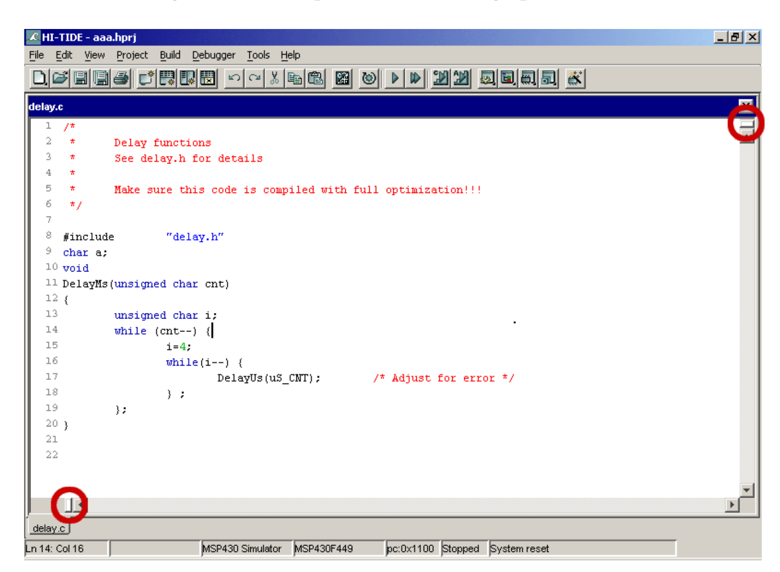
If a file icon is dragged over an region that cannot display the view, the “no drop” mouse pointer is displayed.

2.2.3.2 Focusing Views

A view is in focus if it is showing on the screen and it can accept input from the keyboard. This is indicated by the titlebar of the view being highlighted in a system-dependent colour.

When a view has lost focus, it will be coloured differently (typically grey) to when it was focused. Figure 2.4 shows a Workspace tab with two views inside it. The Editor view on the left is in focus and has its titlebar coloured. The assembly view on the right is not in focus and has a greyed out titlebar.

Figure 2.5: Unsplit view showing split buttons



The colours of the focused and unfocused titlebar is usually user-customizable through the user's desktop settings.

Left-clicking anywhere in the view or on its titlebar will give the view focus and all keyboard input will then be passed to that view. Right-clicking on a view will also give that view focus. With some views, right-clicking will also trigger the view's popup menu.

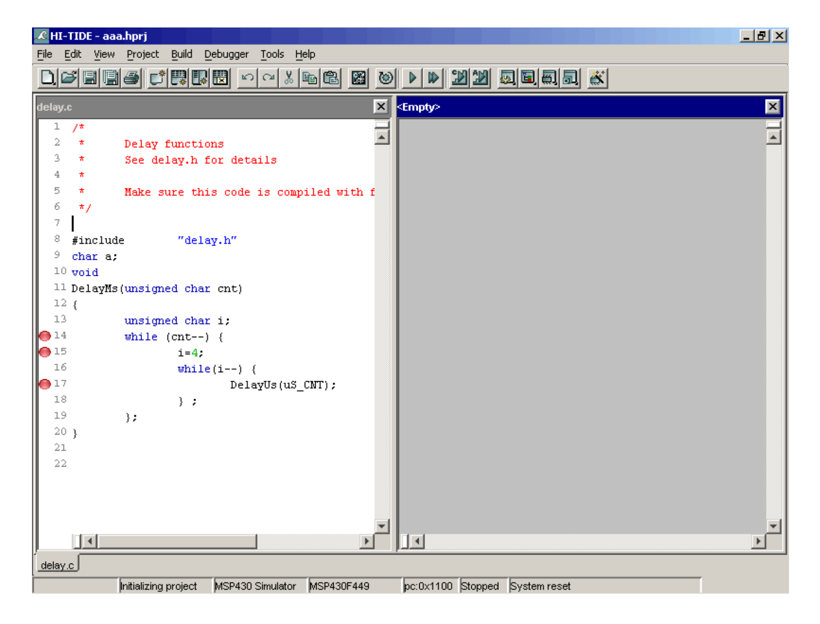
2.2.3.3 Splitting Views

Workspace views can be split into two smaller views, allowing more then one view to be displayed on a Workspace tab at a time. Splitting is performed by a number of ways.

One way is by dragging one of the two split buttons. One split buttons is located at the far left of the horizontal scroll bar; the other is located at the top of the vertical scroll bar in each Workspace view, as shown in Figure 2.5. The view shown has not yet been split.

The split button on the horizontal scroll bar will split the current view into a left and right view, i.e. after the split, there will be two views side by side (see Figure 2.6). The split button on the

Figure 2.6: View split left/right



vertical scroll bar will split the view into a top and bottom view, i.e. after the split, there will be two views one on top of the other (see Figure 2.7). This method of splitting will place the divider between the two views at the location where the user releases the slit button.

Views may also be split by selecting the one of two split view menu items. The **Views** menu has menu items to split the currently focused view left/right or top/bottom: **Split View Left/Right** and **Split View Top/Bottom**, respectively.

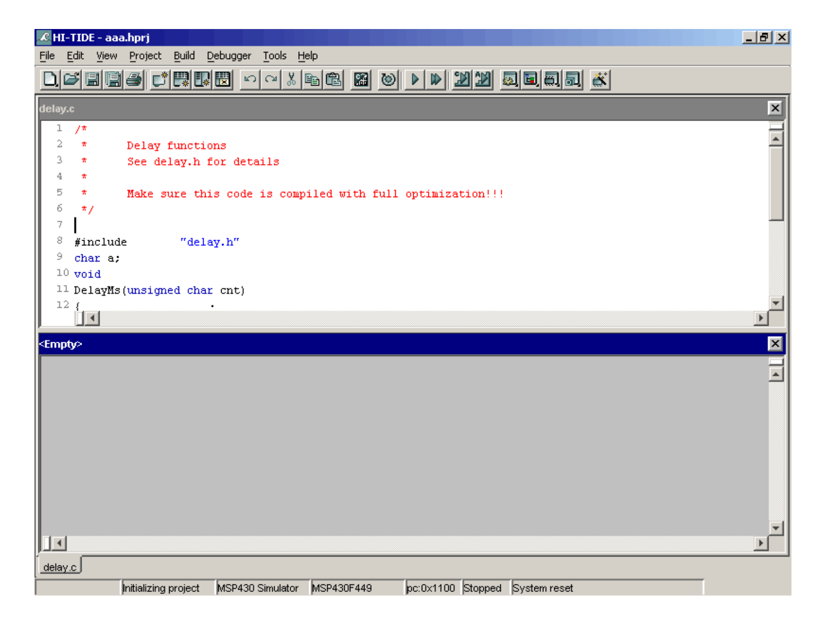
A view can also be split by right-clicking in the view and selecting the splitting of the view options from its popup menu. The popup menu may contain options specific to that view, but will also have the **Split View Left/Right** and **Split View Top/Bottom** options, which will split the views accordingly.

Two toolbar buttons can also be used to split the view in either the horizontal or vertical direction.

After a view has been split, one of the new smaller views will be an *empty view*. The original view is placed either on the left or on the top of the two views, depending on which way the view was split. The empty view can be loaded with another view by dragging a view's toolbar icon as described in 2.2.3.1.

The new views that are formed after splitting can be split themselves, to form more views. This

Figure 2.7: View split top/bottom



process can be repeated until HI-TIDE deems the views too small to be split. This is usually when the scrollbars and split buttons cannot fit into the view's length or width.

2.2.3.4 Closing Views

Each view that can be closed has a close button in the right corner of its titlebar. Clicking the button will close the view. If the view is the only view in a Workspace tab, the tab will be closed as well. If there is more than one view in the tabbed workspace, the view will be closed and the views around the closed view are resized to fill the space that becomes available. Closing a tabbed workspace will close all the views inside it.

The view can also be closed by selecting either the **Close View** menu item in the **Views** menu or the **Close View** button in the standard toolbar.

Right-clicking on a view will display its view-specific popup menu. In addition to the view-specific options, the popup menu will also contain the **Close View** option. Selecting this option will close the view.

A view is also closed when it is replaced by another view. If a view's toolbar is dragged onto an existing view, the view will be closed and replaced with the new view. Refer to Section 2.2.3.1 for more details on creating views from dragging view toolbar icons.

2.2.3.5 View Popup Menu

Right-clicking on Workspace views will display its popup menu. Workspace views may have their own unique menu options, but will always have view-control menu items. The view-control menu items have the following meanings.

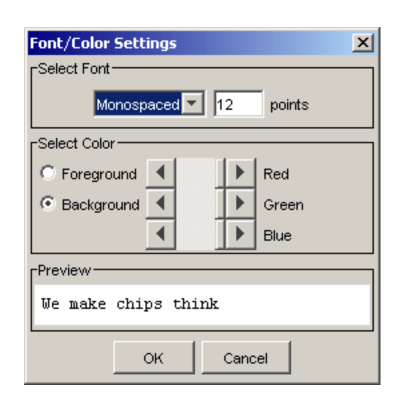
Split View Top/Bottom Selecting this option will split the current view into two views, one on top of the other. The original view will be placed in the top view space and an empty view will be placed in the bottom view space. The two views will be approximately equal in height.

Split View Left/Right Splits the current view into two views, one beside the other. The original view will be placed on the left and an empty view will be added to the right. The two views will be approximately equal in width.

Close View Closes the current view. If the view is the only view in the tabbed workspace, the tabbed workspace will be closed as well. If there are other views in the tabbed workspace, the views will be resized to fill in the space occupied by the closed view.

New Tab Opens and adds a new tabbed workspace in the Workspace area. The new tab will be labelled **New Tab**, but can be renamed. See Section 2.2.2.1 for more details on adding a new tabbed workspace.

Figure 2.8: Font/colour select dialog



2.2.3.6 Changing Font And colour

Most Workspace views allow the font and colours used in its display to be changed. Such views have a **Font/colours...** menu item in their popup menu. Selecting this menu item will display the **Font/colour Settings** dialog, which is shown in 2.8.

There are three sections in the **Font/colour Settings** dialog. At the top is the Select Font area, where the font can be set. The drop-down combo box contains the list of available fonts for that view. To the right of the font selection combo box is the size of the selected font. The exact size can be entered into the text field.

In the center is the colour selection area. The foreground and background font colours can be set, by toggling between the two radio buttons. The three sliders are to adjust the red, green and blue colour values of the font.

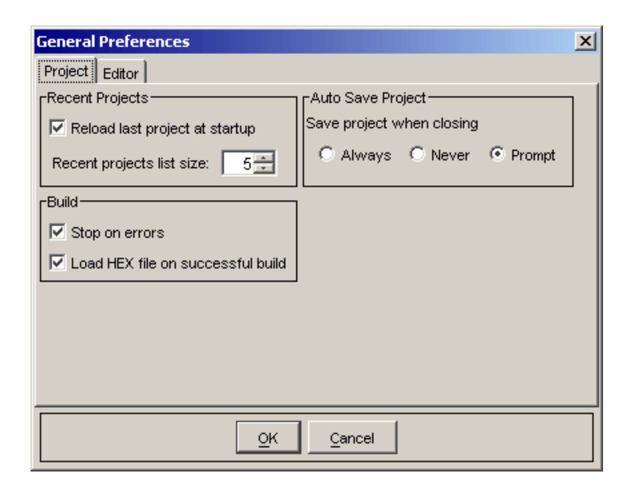
At the bottom of the dialog is a preview of the font type, size and colour that has been set. This is updated as the font properties are changed.

Clicking on **OK** will apply the changes to the view. Clicking **Cancel** will discard all of the changes.

2.3 General Preferences

The general preferences settings allow customization of HI-TIDE's operation. These preferences include text editor functions, through to debugging features. The preferences are set via a **General Preferences** dialog.

Figure 2.9: General preferences dialog — project tab



2.3.1 General Preferences Dialog

The **General Preferences** dialog provides a graphical means of setting options that apply to HI-TIDE and all projects that are used under HI-TIDE.

To display the dialog, select **Preferences** from the **File** menu. The **General Preferences** dialog has tabs to group the options with similar function.

The following sections describe the tabs of the dialog and the function of each of the options.

2.3.1.1 Project Tab

The **Project** tab contains options that apply generally to HI-TIDE projects. The options in this tab are detailed below. Figure 2.9 shows the **General Preferences** dialog with the **Project** tab showing the following options.

Reload last project at startup Selecting this option sets HI-TIDE to automatically reload the last project when HI-TIDE is restarted. De-selecting this option will cause HI-TIDE to start up without any projects opened.

Recent projects list size This option sets the number of file names to display in the recent projects list. These represent projects that have been previously opened by HI-TIDE. This list is displayed under the **Recent Projects** sub-menu of the **Project** menu (see Section 3.1.4). The number of entries must be between 1 and 10, inclusive.

Save project when closing specifies the save action for HI-TIDE to take when a project is being closed. The options are: to always save the project file (**Always**); prompt the user if the project has changed (**Prompt**); or never save the project file (**Never**).

Stop on errors specifies whether compilation should continue if an error is encountered. Prematurely stopping compilation may save time compiling large projects. De-selecting this option will force the compiler to compile all of the source files before reporting any errors. Use this setting if you want the compiler to report all errors with all files. The link step is never performed is there are any errors in the source files, regardless of this setting.

•

Often one error may causes other errors to occur in a snow-ball effect. Primarily concentrate on the first error(s) issued by the compiler. Seemingly extraneous error messages may disappear after fixing problems earlier in the source and the project is recompiled.

Load HEX file on successful build If this option is selected, HI-TIDE will attempt to load the HEX file into the debugger if the project was successfully built. This ensures that the debugger is always working with the latest build output. When deselected, the user must manually load the HEX file, if required, after a successful build.

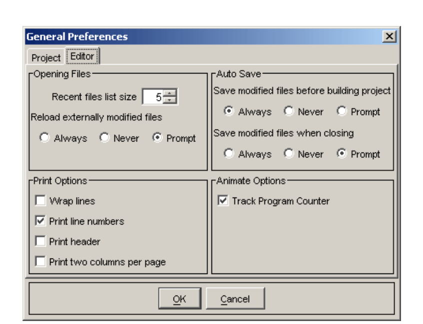
2.3.1.2 Editor Tab

The **Editor** tab sets options specific to the editor and editing of files. The options are described in the following. Figure 2.10 shows the **General Preferences** dialog with the **Editor** tab showing the following options.

Recent files list size This option sets the number of file names to display in the recent files list. These represent files that have been previously opened in an Editor view. The list is displayed under the **Recent Files** sub-menu of the **File** menu (see Section 4.16). The number of entries must be between 1 and 10, inclusive.

Reload modified files This option allows control over the editor's automatic load feature. If a file shown in an Editor view has been modified other than by the editor, it can be reloaded to display the changes. The editor will check all files if HI-TIDE regains focus from another application, or after compilation of any files. The options are to always reload files if they have been modified (**Always**), prompt the user to reload the files if they have changed (**Prompt**) or never reload the modified files (**Never**). If a file has been modified, but the user opts not to update the Editor view, the normal **file modified** flag will appear next to the file's name in the title bar of the Editor view.

Figure 2.10: General preferences dialog – editor



Save modified files before building This option allows control over the editor's automatic save feature. Prior to building the project, the editor can check to ensure that each file to be compiled has been saved. If the user opts not to save modified files, the file built during compilation will be the file without the modifications, i.e. the file as it is stored on disc. The settings associated with this option are to always save the files prior to building (**Always**), prompt if the files need saving (**Prompt**) or never save modified files prior to building (**Never**).

This option also applies to files compiled individually from the **Compile To** menu under the **C File** popup menu or **Assembler file** popup menu.

Save modified files when closing This option allows control over the editor's automatic save feature. Prior to closing, the editor checks to see if any opened files have been modified. If the user opts not to save modified files when closing, any changes made in the editor will be lost. The settings associated with this option are to always automatically save unsaved modified files (**Always**), to prompt the user if the file needs saving (**Prompt**) or to not save the files at all (**Never**).

Wrap lines This option enables line wrapping when printing files from the editor. It is recommended that this option be selected to ensure that long lines are not truncated. This is critical when using the **Print two columns per page** option. This option does not affect how lines are displayed in the Editor view.

Print line numbers This option controls whether line numbers are printed with the file. When selected, a line number will be printed at the beginning of each line of code.

Print header This option controls whether a header is printed with the file. When selected, a header containing the filename and current date, and a footer containing a page number, are added to

each page in the file.

Print two columns per page This option allows printing of two columns on one page with landscape layout. To prevent text being printed off the column and page, select the **Wrap lines** option when using selecting this option.

Track Program Counter This option controls how the editor behaves when stepping through a program. When this option is enabled, the Editor view will track the program counter and adjust its display so as the line about to be executed is visible on the screen. As the program calls or jumps to code in different files, HI-TIDE will automatically switch views to display the file in which the PC is located. If execution is transfered to a file that is not open in any editor, that file will be opened.

2.4 Third-Party Tools

Third-party software tools can be executed without having to leave the HI-TIDE environment. This feature allows customized buttons to be added to the HI-TIDE toolbar which can be used to launch the associated third-party software. These buttons may also be used to switch to the third-party tool once they are running.

Third-party tools, when added, are shown in the buttons list in the **Setup Third-Party Tools** dialog and are represented by buttons in HI-TIDE's *User Tools* toolbar (see Section 3.2 for more information on toolbars).

Some tools are automatically added by HI-TIDE and can not be removed from the tool buttons list, although they may be hidden from the toolbar.

The third-party tools feature is mainly controlled via the **Setup Third-Party Tools** dialog. This dialog can be opened from the **Tools** menu and appears as shown in Figure 2.11. It can be used to add, remove and customize tools. The **Setup Third-Party Tools** dialog can also be used to hide, modify or remove the buttons once they have been installed on the toolbar.

At the top of the dialog is a list of the third-party tools buttons that are currently setup in HI-TIDE. The center section of the dialog shows the details associated with each button. Selecting a button from the list will enable the details panel and the details of the button will be displayed in the corresponding text fields.

2.4.1 Adding and Deleting Tools

To add a new third-party tool to HI-TIDE, click on the **New Tool** button in the tool setup dialog. This will create a new entry and the entry is added to the list of tools. The entry will be labelled **Untitled*n***, where ***n*** is a sequential index, starting from 0 which is incremented each time a new tool button is created. The tool's name may be changed from the default name to better describe the tool.

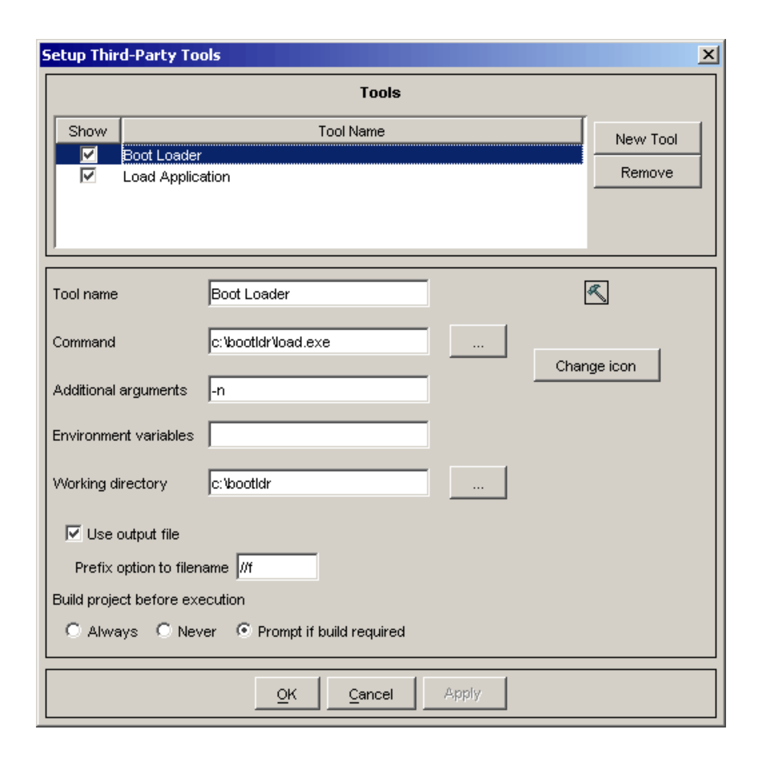


Figure 2.11: Tool setup dialog

All third-party tool buttons must have unique names. These names are used to identify the buttons. A newly created tool button cannot have the same name as an existing one.

The new tool button entry is assigned a default "hammer" button image. The button image can be changed to a customized image. Changing of the button image is discussed in Section 2.4.2. The other options are described in this section as well.

When all details have been entered, click either the **Apply** button or the **OK** button to save the new button. The **Apply** button saves the changes, but does not close the dialog. Clicking on **Cancel** will discard any unsaved changes. When the dialog is closed, new tools will be displayed in the HI-TIDE toolbar.

To remove a button from the list, select the button in the list and click on **Remove**. Only user-created buttons can be removed from the list. Default buttons created by HI-TIDE can not be removed, but they can be hidden. For more information on hiding buttons, refer to Section 2.4.3.

2.4.2 Tool Options

Adding a new tool is described in Section 2.4.1 and once added, the following describes the different text fields and options within the details panel of this dialog that apply to each tool. These options may be edited after a tool has been setup by re-opening this dialog and making the appropriate changes.

Tool name A text field for entering the name of the third-party tool button. This name may be any name to describe the tool, but it must be unique within the list of tools.

Command This is the actual command that is executed by HI-TIDE to start the third party software. There is a widget button to the right of this text field that opens a file dialog to select an executable. The *full path* of the executable should be entered here. Do not enter command-line arguments in this field.

Additional arguments These are command-line arguments that will be passed to the executable. Leave this field blank if there are no arguments required. Do not add the name of the compiler's output file here. That can be automatically added by HI-TIDE. See the **Use output file** option, described below. The arguments required and their format is specified by the application being executed. See the documentation that came with this software for more information.

Environment variables Any required environment variables can be entered into this text field. Leave this field blank if there are no environment variables required. The environment variables required and their format is specified by the application being executed. See the documentation that came with this software for more information.

Working directory This text field specifies the working directory for the executable. This directory is where the command will be executed from. A widget to the right of the text field opens a

directory chooser to help select the required directory. Leave this field blank if not working directory need be specified.

Change icon Clicking on this button opens a file chooser to allow the button image to be specified. A preview of the tool button image is shown above the **Change icon** button. If no image is selected, a plain grey button is used on the toolbar. A generic image, `hammer16.gif`, can be loaded from the `images` directory under HI-TIDE.

Use output file This option allows the name of the compiler's output file to be specified to the third-party tool. Selecting this option passes the name of the compiler's output file as an argument to the executable command when it is executed. If the name of the output file changes, this feature will always choose the current output file name.

When using this feature, the **Prefix option to filename** text field will be enabled. This field specifies any command-line arguments that need to be prefixed with the filename. Leave this field blank if the filename does not need a leading command-line argument before it.

Build project before execution This option controls the auto-build feature of HI-TIDE. After clicking on a third-party tool button, HI-TIDE checks the status of the project prior to launching the third-party tool. If **Always** is selected, then the project is automatically built each time. Selecting **Never** will immediately launch the third-party tool without building the project, even if the project is out of date. If **Prompt** is selected the user will be prompted to select the required action.

In all cases, the project is built as if the **Build** button or menu item was selected, and uses dependency checking. Thus, if a build is requested, but no files have changed, no actual compilation will take place.

OK Button Clicking on the **OK** button will save the current dialog settings and close the dialog.

Cancel Button Clicking on the **Cancel** button will close the dialog and any unsaved changes will be lost.

Apply Button Clicking on the **Apply** button will save the current dialog's settings. Once the details have been saved, the details are not discarded when the **Cancel** button is clicked.

2.4.3 Hiding and Showing Buttons

The third-party tool buttons can be removed from the toolbar without being deleted entirely from the list of tools.

•

If a tool button should be showing on the toolbar, but is not present, it may be that the toolbar is hidden. Refer to Section 3.2.1 for more details on displaying the Tools tool bar.

All the created tool buttons are listed in the tool buttons list. The list contains two columns: The first column is the **Show** column which contains checkboxes. The second column contains the **Tool Name**, which displays the names assigned to each button. Deselecting the check box in the show column for a button will remove that button from the HI-TIDE toolbar, but will not remove the button from the list. Selecting the checkbox will display the button in the HI-TIDE toolbar.

Changes to hiding or showing the buttons will only apply if the **OK** button or **Apply** button in the dialog is clicked.

Chapter 3

HI-TIDE Menus and Toolbars

3.1 Menus

This section presents a description of each of the HI-TIDE menus.

3.1.1 File Menu

The **File** menu contains menu items that relate to files used by HI-TIDE. To conform with other applications, this menu also contains a preferences item and an exit menu item. The following describe the menu items in detail.

New File This will create a new text editor file. Each new file is opened in a new Workspace tab. As new files are created, they will be named **Untitled*n*** where ***n*** is a sequential number. The Workspace's tab in the editor will be labelled with the same name as the file.

Open... Opens a file dialog to select a file to load in the editor. The selected file will be opened in a new editor Workspace tab. The Workspace's tab in the editor will be labelled with the same name as the file.

Recent Files A sub-menu that lists the files that have been recently opened in HI-TIDE. The size of the list can be set in the editor options of the **General Preferences** dialog, see Section 2.3.1.2. The selected file will be opened in a new Workspace tab. The tab will be labelled the same as the name of the file.

Save File Saves the opened editor file that is currently in focus. If the currently focused view is not of an editor file this action will have no effect. If the file is a new file that has not yet been

saved with a user-specified name, the user will be prompted to enter a name with which to save the file as.

Save All Saves all of the currently opened editor files and the project file. Untitled files will be saved in the same manner as **Save File As**.

Save File As... Opens a file dialog to allow entry of a new file name with which to save the current file. This will rename the opened file.

Print... Displays the **Print** dialog for printing of an editor file. Only editor views can be printed. This menu item will only open the **Print** dialog if an editor view was focused prior to selecting this option. Selecting this option while an editor view is not in focus will not have any effect. See Section 4.3.6 for more details on printing editor files.

General Preferences... Opens the **General Preferences** dialog for selecting preferences that apply to HI-TIDE and other components of HI-TIDE.

Exit Close and exit from HI-TIDE. If there is a project currently opened, the user may be prompted to save the project, as dictated by the general preferences settings described in Section 2.3.1.1.

3.1.2 Edit Menu

The **Edit** menu provides functions relating to the text editor. The functions are listed and described as follows.

Undo Reverses the last text editor action. Several editor actions can be reversed by using this item repeatedly.

Redo Restores the edit action that was removed by the last **Undo** action.

Cut Copies the currently selected text in a text editor view to the clipboard and then deletes the selected text.

Copy Copies the currently selected text to the clipboard. The selected text is not deleted.

Paste Inserts the contents of the clipboard into the selected editor file, before the current position of the cursor.

Find... Opens the **Find & Replace** dialog with the **Find** tab selected. This allows the user to search for text within the currently selected editor view. An editor view must be selected for this menu item to have an effect. See Section 4.8 for a detailed description of the **Find & Replace** dialog.

Find Again Repeats the last search without displaying the **Find & Replace** dialog. An editor view must be selected for this menu item to have an effect.

Replace... Opens the **Find & Replace** dialog with the **Replace** tab selected. This allows the user to search and replace text within the currently selected editor view. An editor view must be selected for this menu item to have an effect. See Section 4.9 for a detailed description of the **Find & Replace** dialog.

3.1.3 View Menu

The **View** menu provides actions that relate to some of the visual components within HI-TIDE. The menu items are described below.

Toolbar The **Toolbar** menu is a submenu that shows a list of the toolbars available in HI-TIDE. The user is able to select which toolbars will be displayed in HI-TIDE by selecting the appropriate item in the **Toolbar** menu. The toolbars which are displayed are marked with a check.

Split View Top/Bottom Splits the currently selected Workspace view into two views. The result will be two views, one on top of the other. The original view will be placed in the new top view. The bottom view will be an Empty view. See Section 2.2.3.3 for more details on splitting of views. Selecting this option while no views are in focus will have no effect.

Split View Left/Right Splits the currently selected Workspace view into two views. The result will be two views side by side. The original view will be placed in the new left view. The right-hand side view will be an Empty view. See Section 2.2.3.3 for more details on splitting of views. If a view is not in focus when selecting this option, it will have no effect.

Close View Closes the currently focused Workspace view. If this Workspace view is the only view in the Workspace tab, the tab is removed as well. If the Project area or Build area are in focus when this option is selected, they are hidden from display . For more information on the Workspace area, refer to Section 2.3.

New Tab Adds a new tab to the Workspace area. The new tab will be labelled **New Tab** and will contain an Empty view. See Section 2.2.2.1 for more details.

Show/Hide Project area Shows or hides the Project area. Refer to Section 4.1 for more details on the Project area.

Show/Hide Build area Shows or hides the Build area. Refer to Section 4.2 for more details on the Build area.

The following **View** menu items create new views which are then added in new tabs in the Workspace area. As the views are plugins, the order in which they appear in the menu may differ. The views are described in alphabetical order.

Data Memory Displays the writable memory of the target device. See Section 4.4.2 for more details.

Executable Memory Displays the executable memory of the target device. See Section 4.4.1 for more details.

Registers Displays the registers memory of the target device. See Section 4.4.3 for more details.

Variable Watch Displays the view to monitor variables and their values. This view can be used to display any variables defined by a program. See Section 4.4.4 for more details.

Local Watch Displays the view to monitor block-scope variables and their values. This view automatically populates with `auto` and static local variables defined within the function being executed. See Section 4.4.4 for more details.

Virtual I/O Displays a view in which peripheral device panels which can be added and wired to the simulator.

3.1.4 Project Menu

The **Project** menu contains project-related menu items. These are described below.

New Project... Opens the Project wizard which will help create a new HI-TIDE project.

Open Project... Opens a file dialog for selection of an existing project to load.

Recent Projects A submenu which displays a history list of projects that have been opened previously. The number of projects in the history can be set in the **General Preferences** dialog, see Section 2.3.1.1. Selecting a project from this submenu will reopen the project if the project file still exists.

Close Project Closes the currently opened project. Options in the **General Preferences** dialog can be set to determine if the project file is to be saved before closing, see Section 2.3.1.1. Similar options also apply for unsaved Editor view files that are being closed.

Save Project Saves the currently opened project file to disk.

Save Project As... Saves the currently opened project under a different name or path. A file dialog will appear for the user to select the new name and path.

Add Files To Project... Opens a file dialog for the user to select files to add to the project. The files that can be added to the project include source files (`.c` or `.as`), object files (`.obj`) and library files (`.lib`). Multiple file selection can be performed in the file dialog.

Add File To Project Adds the currently opened and focused Editor view file to the project. If the file cannot be added to the project, this option is disabled.

Change Toolsuite Version... Selects the toolsuite version to use with the project. The toolsuite relates to the compiler and compiler version. For more details on toolsuites, refer to Section 5.1. For more details on changing toolsuites, refer to Section 5.6.1.

Change Device... Selects the target device for the project. A chip selection dialog is shown to enable the selection of the new microcontroller. This action can also be performed by double-clicking on the target name in the status bar. Section 5.6.2 has more details on changing target devices.

Change Package... Selects the package type of the target device. A chip package selection dialog will appear to enable the selection of the new chip package. Some microcontrollers have functionality that is dependent on the package type. See Section 5.6.3 for more information on Variable Watch view package types.

Change Debugger... Selects the debugger to use with the project. The debuggers available will depend on the toolsuite selected. This action can also be performed by double-clicking on the debugger name in the status bar. For more details on changing debuggers, see Section 5.6.4.

Global Compiler Options... Opens the **Global Compiler Options** dialog to allow setting of global compiler options. Global compiler options affect all the files in the project. The dialog and options displayed by this menu are very much dependent on the toolsuite selected. The compiler options are discussed in detail in Chapter 7. The **Global Compiler Options** dialog can also be displayed by double-clicking on the output node in the Files view in the Project area.

3.1.5 Build Menu

The **Build** menu contains the actions to build the current project. The options are:

Make Builds the project with dependency checking. Only source files that are not up to date are recompiled. If the output node is not up to date, the object files are linked to create an updated output node.

Make All Compiles all of the source files in the project and then links to create the output node.This action always recompiles each source file and relinks the project even if the files have not been modified.

Clean Deletes all compiler-generated files, e.g. object files (`.obj`), list files (`.lst`), source-level debugging files (`.sdb`, `.sym`), etc. Object files that are specified in the project are not removed.

Compile To Object File Compiles the current focused file showing in the Editor view to an object file. This option is only enabled if the view in focus is an editor view. No other files are compiled and the project is not linked. This option can be used to locate errors within the file currently being developed.

3.1.6 Debugger Menu

The menu items in the **Debugger** menu are to control the selected debugger. If no debugger is selected, this menu has no effect. The active debugger is displayed in the status bar. The menu items are described in the following.

Reset Performs a reset of the debugger. Refer to the documentation of the debugger for more details on what the reset action does at the debugger level.

Run Commences full-speed execution of the program in the debugger. Debugger views will not be updated during full-speed execution. To stop this action, use the **Stop** item.

Animate Continuously executes single assembler steps, updating debugger views after each step. Execution is slower than that associated with the **Run** action. To stop the debugger, use **Stop**.

Stop Stops the current execution of the debugger.

C Step Makes the debugger execute a series of assembler instructions which correspond to one line of C source code. The number of executed instructions will depend on the C source statement. Debugger views are updated when the debugger has executed the instructions.

Assembler Step Makes the debugger execute a single assembler instruction. Debugger views are updated once the debugger has executed the instruction.

Set/Remove Breakpoint Allows a breakpoint to be set or removed on the C line which contains the caret in a focused Editor view, or on the highlighted assembler line in a focused Disassembly view. A set breakpoint is indicated by a red dot. The dot disappears if no breakpoint is set on this line. See Section 9.1.2 for more information.

Disable Breakpoint Disables, but does not remove, a breakpoint from the C line which contains the caret in a focused Editor view, or on the highlighted assembler line in a focused Disassembly view. The disabled breakpoint is indicated by a grey dot.

Remove All Breakpoints Removes all breakpoints from the program, whether they were set on C statements or assembler instructions.

Disable All Breakpoints Disables, but does not remove, all breakpoints from the program, whether they were set on C statements or assembler instructions.

Enable All Breakpoints Enables all disabled breakpoints from the program.

Run To Cursor Causes a temporary breakpoint to be inserted on the C line which contains the caret in a focused Editor view, or on the highlighted assembler line in a focused Disassembly view and program execution to continue from the current program counter location.

Load HEX File... Opens a file dialog to select the HEX file to load into the debugger memory. This can be used to over-write a HEX file previously loaded into memory. The HEX file can be automatically loaded after building. See Section 2.3.1.1.

3.1.7 Tools Menu

The **Tools** menu provides functions to access external or third party tools, as well as the Code Wizard.

Code Wizard Opens the **Code Wizard** dialog. The Code Wizard is used to aid in the initialization of target device peripherals. See Chapter 6 for more information.

Setup User Tools... Opens the **Setup Third-Party Tools** dialog. See Section 2.4 for more details on user tools.

3.1.8 Help Menu

Contains various help information.

About Shows the **About** dialog, listing details of HI-TIDE, including the version number, copyright, contact and trademark information.

3.2 Toolbars

This section presents an item-by-item description of each of HI-TIDE's toolbars and the functions of each of the toolbar buttons.

3.2.1 Hiding / Showing Toolbars

HI-TIDE's toolbars can be hidden from view so that they do not clutter the toolbar display.

To hide a toolbar, select the **View** menu, and then the **Toolbar** sub-menu. This will display a list of the available toolbars. The toolbars that are currently visible will be marked with a check next to

their name. Those without the check are hidden. To hide a toolbar that is showing, select that toolbar name from the **Toolbar** sub-menu.

To show the toolbar, select the toolbar name from the same menu.

3.2.2 Standard Tools Toolbar

The Standard Tools toolbar provides standard tools such as creating new files, saving files, printing and splitting of views. The standard toolbar buttons are listed and described below.

New File This will create a new editor text file called **Untitled*n***, where ***n*** is a sequential number. This is the same as selecting **New File** menu item in the **File** menu. The **New File** toolbar button can be “dragged and dropped” to an Editor view create a new file. Refer to Section 4.3.2 for more details.

Open File Opens a file dialog for selection of an existing text file to open in the editor. This is the same as selecting the **Open File** menu item in the **File** menu.

Save File Saves the currently focused file in the editor. This is the same as selecting the **Save File** option in the **File** menu.

Save All Saves the currently focused file in the editor. This is the same as selecting **Save All** in the **File** menu.

Print Opens the **Print** dialog for setting up print options and printing an editor file. This is the same as selecting **Print...** from the **File** menu.

New Tab Creates and adds a new Workspace tab to the Workspace area. This is the same as selecting the **New Tab** menu item in the **View** menu.

Split View Top/Bottom Splits the currently focused view into two views, one on top of the other. This is the same as selecting **Split View Top/Bottom** in the **View** menu.

Split View Left/Right Splits the currently focused view into two views, side by side. This is the same as selecting **Split View Left/Right** in the **View** menu.

Close View Closes the focused view. The is the same as selecting **Close View** in the **View** menu.

3.2.3 Editor Toolbar

The Editor toolbar provides tools essential to the editor. These include actions such as cut, copy and paste. The editor toolbar buttons are listed and described below.

Undo Reverses the last editor text action(s). This has the same effect as selecting **Undo** from the **Edit** menu.

Redo Restores the edit action that was reversed by the last **Undo** action. This is the same as selecting **Redo** from the **Edit** menu.



Cut Copies the selected text from the editor text file to the clipboard and then deletes that selected text from the file. This is the same action as selecting **Cut** from the **Edit** menu.



Copy Copies the selected text from the editor text file to the clipboard. This is the same as selecting **Copy** from the **Edit** menu.

Paste Inserts the contents of the clipboard into the editor text file before the current position of the cursor. This is the same as selecting **Paste** from the **Edit** menu.



3.2.4 Build Toolbar

The Build toolbar provides tools related to project compilation. The build toolbar buttons are listed and described below.

Make Project Builds the current project files, using dependency checking. This is the same as selecting the **Make** option in the **Build** menu



3.2.5 Views Toolbar

The Views toolbar provides means of creating and adding new views to the Workspace area. The buttons in the Views toolbar can be "dragged and dropped" into the Workspace area to create new views. Clicking on the view buttons will create a new view in a new Workspace tab. Dragging the button and dropping the view will replace the current view with the dragged view. See Section 2.2.3 for more information.

The Views toolbar buttons are listed and described below.

Data Memory Creates a new view containing the writable memory view of the target device. This is the same as selecting **Data Memory** from the **View** menu. This button can also be dragged and dropped to create a new view.



Executable Memory Creates a new view of the of executable memory of the target device. This is the same as selecting the **Executable Memory** menu item from the **View** menu. This button can also be dragged and dropped to create a new view.



Registers Creates a new view of the registers of the target device. This is the same as selecting Registers from the view menu. This button can also be dragged and dropped to create a new view.



Watch Variables Displays the view to monitor variables and their values. This view can be used to display any variables defined by a program. This is the same as selecting the **Watch Variables** menu item from the **View** menu. This button can also be dragged and dropped to create a new view.

Local Watch Displays the view to monitor block-scope variables and their values. This view automatically populates with `auto` and static local variables defined within the function being executed. See Section 4.4.4 for more details.

Virtual I/O Displays a view in which peripheral device panels which can be added and wired to the simulator.

3.2.6 Tools Toolbar

The Tools toolbar contains any tools that are provided with HI-TIDE. The buttons in this toolbar are listed and described below.

Code Wizard Launches the **Code Wizard** dialog. See Chapter 6 for more information.

3.2.7 User Tools Toolbar

The User Tools toolbar contains user-defined tools and actions. The buttons are customizable and may contain different icons. See Section 2.4 for more details on setting up and using third party tools.

3.2.8 Debugger Toolbar

The Debugger toolbar buttons are used to control the debugger that is selected. The buttons are disabled if there is no debugger selected for the current project.

The basic functions of the debugger are controlled by these toolbar buttons, which are listed and described below.

Reset Resets the debugger. The extent of the reset will be dependent on the debugger. Refer to the debugger's documentation for more details on what type of reset the debugger performs. This button has the same effect as selecting **Reset** in the **Debugger** menu.

Run Does a full speed execution of the code in the debugger. Debugger views are not updated while the debugger is running at full speed. While the debugger is running, this button is replaced with the **Stop** button, which is used to terminate the full speed execution of the debugger. Clicking on this button has the same effect as selecting **Run** in the **Debugger** menu. This button is disabled while another action such as **Animate, C Step** or **Assembler Step** is executing.

Animate Continuously executes single assembler steps, updating debugger views after each step. This has the same effect as continuously clicking on the assembler step button. **Debugger** views are updated while this action is running. While the debugger is running, the **Animate** button is replaced by the **Stop** button. The debugger views are updated at the end of each execution each assembler instruction. This action can also be performed by selecting the **Animate** option from the **Debugger** menu. This button will be disabled while the debugger is operating under **Run**, **C Step** or **Assembler** Step.



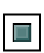
C Step Steps the debugger a series of assembler instructions equivalent to one line of C source code. While the debugger is stepping, this button is replaced with the **Stop** button and is reinstated after the stepping stops. Selecting this button is the same as selecting **C Step** from the **Debugger** menu. The debugger views are not updated while the debugger is stepping, but are updated once the stepping is completed. This button will be disabled while other actions such as **Run**, **Animate** or **Assembler Step** are being executed by the debugger.



Assembler Step Makes the debugger execute one assembler instruction. The debugger views are updated at the end of the execution of the assembler step. This is the same as selecting the **Assembler Step** option from the **Debugger** menu. This button will be disabled while the debugger is operating under **Run**, **Animate**, or **C Step**.



Stop This button will appear in place of the **Run** button, **C Step** button or **Animate** button, when one of those actions have been selected. The **Stop** button will appear in lieu of the button that started the debugger. Selecting this button will stop the debugger, and the original button will be again displayed.



3.3 The Status Bar

The Status Bar can contains information relating to the Editor view, the currently selected debugger, the selected target device, and information relating to the execution of a program being debugged. A typical status bar display is shown in Figure 3.1.

On the far left of the status bar is the Editor view;s caret position indicator. This indicator shows the line number and column position of the caret, in relation to the file in the Editor view that is currently being edited. If an Editor view is not in focus, no line or column information will be displayed.

To the right of the caret position indicator is the editor status. This reports events in the editor. The events are reported as text messages. Typical events can include if a search in the editor has reached the end of file, etc.

To the right of the editor status is a description of the debugger that is currently selected. This will display either the name of the debugger that is selected or the text **No Debugger** when a debugger

Figure 3.1: The status bar



is not selected. Double clicking on this section will display a dialog to allow selection of a different debugger, if a different debugger is available.

The name of the project target device is displayed in the third section of the Status Bar. Double clicking on this section will display a dialog to allow selection of a different target device.

The fourth section contains the value of the program counter. If no debugger is selected this section will be blank.

The next section after the program counter indicates the status of the debugger. The debugger can be in one of three states and is indicated by the text: **Stopped**, **Busy** or **Running**. Stopped indicates that the debugger has halted and is not performing any tasks. Busy indicates that the debugger is performing some task that does not involve program execution. Example of a busy state for a debugger could be resetting or downloading memory. The Running state indicates that the debugger is executing the program.

A description of why the debugger last stopped is shown in the far right section of the Status Bar. On most occasions this section will display **User Requested** or **Breakpoint** as the reason execution was stopped. On some occasions there will be an error in executing the program causing the debugger to stop, an explanation of the error will be displayed in this section.

Chapter 4

HI-TIDE Views

This chapter looks at all the views which can be displayed in HI-TIDE.

4.1 The Project Views

The views within the Project area are static views that are always visible and which are managed by HI-TIDE. The following sections describe their purpose and options.

4.1.1 Files View

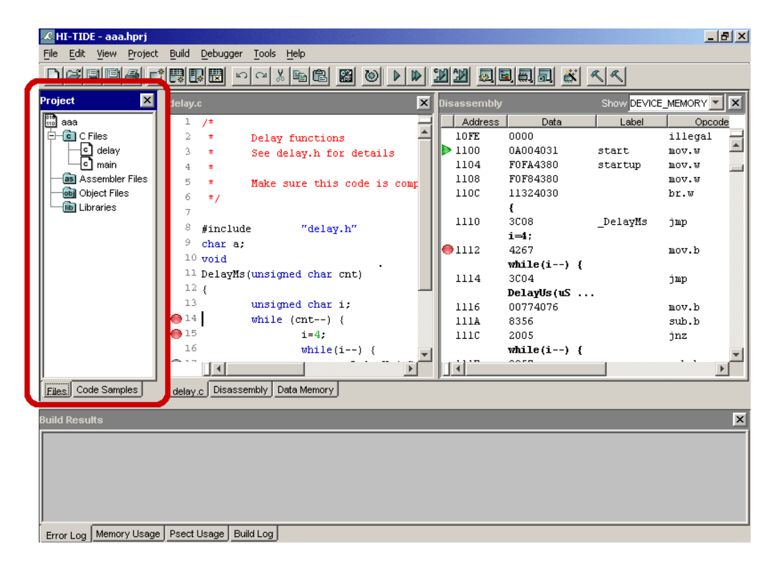
The Files view is one of two views within the Project area and displays the files associated with the application being developed. It is shown circled in Figure 4.1.

The files are displayed like a directory structure. At the top (root) of the structure is the output file (or output node). The next level under this root are the file folders. The folders contain the source files, object files and libraries associated with the project.

The folders are **C Files** (which contains C source files), **Assembler Files** (which contains assembler source files), **Object Files** (which contains user-supplied relocatable object files) and **Libraries** (which contain user-supplied, HI-TECH-format library files).

By right-clicking on the files or folders a popup menu will be displayed showing various options. These options include functions such as adding files to the project, creating new files, opening the file in the editor, etc. The following sections describe these options in detail.

Figure 4.1: Project area



4.1.1.1 Output File Popup Menu

Double-clicking on the output file node will open the **Global Compiler Options** dialog for the project. Right-clicking on the output file node provides the following options in a popup menu.

Global Compiler Options... The **Global Compiler Options...** command opens the **Global Compiler Options** dialog for the project. The dialog allows the setting of project-related options, on a global scale to the project. These options include compiler options, memory options and linker options. These options are discussed in Chapter 7.

Properties The **Properties** command opens the **File Properties** dialog, which displays properties of the default output file stored on disk. See Section 4.1.2 for more information on the **File Properties** dialog.

4.1.1.2 C Files Folder Popup Menu

Right-clicking on the **C Files** folder provides the following options.

Add Existing C File(s) This command allows one or more existing C files to be added to the project. The file(s) to be added to the project can be selected via the file dialog that is shown.

Create And Add New C File This command will show the file dialog which will prompt for a file name and path. Once a file name is entered, the new file, with the specified name, will be added to the project in the appropriate file folder and will be opened in the editor as a new file.

4.1.1.3 C File Popup Menu

Double clicking on a file in the **C Files** folder will open that file in the editor. If the file is currently opened in the project, the editor will bring the view with that file into focus. If the file is not opened in the editor, the file will be opened and placed in a new Workspace tab, and the tab will be labelled with the name of the file.

Right-clicking on the individual C file will select that file and show a popup menu with the following options:

Remove from project This option removes the selected file from the project.

File-specific options... The **File-specific options...** command allows the setting of compiler options which only apply to the selected file. Any options set for a single file will override the global options specified for the project. The file-specific options which can be specified will be dependent on the compiler selected.

Compile to... This menu contains commands to compile the selected C file to the following formats

Preprocessed file This command compiles the selected file to a pre-processed file (`.pre`). The output file will be the name of the C file but will have a `.pre` extension.

Assembler file This command compiles the selected file to an assembler file (`.as`). The output file will be the name of the C file but will have a `.as` extension.

Object file This option compiles the selected file to a re-locatable object file (`.obj`). The output file will be the name of the C file but will have a `.obj` extension.

Properties... The **Properties...** command opens a dialog that displays properties of the selected file stored on disk. See Section 4.1.2 for more information on the **File Properties** dialog.

4.1.1.4 Assembler Files Folder Popup Menu

Right-clicking on the **Assembler Files** folder provides the following options:

Add Existing Assembler File(s) This command allows one or more existing assembler files to be added to the project.

Create And Add New Assembler File This command will prompt for a file name and path. Once a file name is entered, the new file with the specified name will be added to the project and is also opened in the editor as a new file.

4.1.1.5 Assembler File Popup Menu

Double clicking on the file will open the file in the editor. If the file was not already opened in the editor, the file will be opened and placed in a new Workspace tab, with the tab labelled with the name of the file. If the file was already opened, the editor will bring that file into focus.

Right-clicking on the individual assembler file will select that file and show a popup menu with the following options:

Remove from project This option removes the file from the current project.

File-specific options... The **File-specific options...** command allows the setting of compiler options which only apply to the selected file. Any options set for a single file will override the global options specified for the project. The file-specific options which can be set will be dependent on the compiler used.

Compile to... The **Compile to...** menu contains commands to compile the selected assembler file to the following formats

Preprocessed file This command compiles the selected file to a pre-processed file (`.pre`). The resulting file will be the name of the C file but will have a `.pre` extension.

Object file This option compiles the selected file to a relocatable object file (`.obj`). The resulting file will be the name of the C file but will have a `.obj` extension.

Properties... The **Properties...** command opens a dialog that displays properties of the file stored on disk. See Section 4.1.2 for more information on the **File Properties** dialog.

4.1.1.6 Object Files Folder

Right-clicking on the **Object Files** folder will show a popup menu with the following options.

Add Existing Object File(s) This command allows additional pre-compiled object files to be added to the project. D not add object files created from C or assembler project files.

4.1.1.7 Object Files

Right-clicking on the **Object Files** will select that file and show a popup menu with the following options.

Remove From Project The **Remove From Project** command will remove the object file from the project. If the object file is the standard object file, confirmation will be required before the file is removed as this is not a common operation.

Properties... The **Properties...** command opens a dialog that displays properties of the file stored on disk. These properties are the absolute path of the file, the file's size in bytes and the time and date it was last modified. See Section 4.1.2 for more information on the **File Properties** dialog.

4.1.1.8 Libraries Folder

Right-clicking on the **Libraries** folder shows a popup menu with the following options.

Import Library This command allows the addition of pre-compiled libraries to be added to the project.

4.1.1.9 Library Files

Right clicking on the library file selects that file and provides the following options.

Properties... The **Properties...** command opens a dialog that displays properties of the selected file stored on disk. See Section 4.1.2 for more information on the **File Properties** dialog.

Remove From Project The **Remove From Project** command will remove the selected library file from the project. If the library file is a standard (compiler supplied) library file, confirmation will be required before the file is removed as this is not a common operation.

4.1.2 File Properties Dialog

The File Properties dialog displays the properties of a file stored on disk. The information includes the name of the file, the full directory path of the file, the size of the file and when it was last modified.

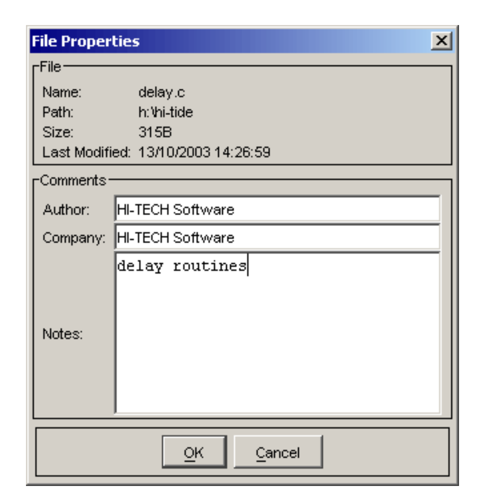
Additional information can also be associated with the file. This information includes the author of the file, the company and any accompanying notes. The additional information are text saved with the file and can optionally be filled in. This information is not actually saved with the file but with the project information and will not affect the file in any way. Figure 4.2 shows a typical **File Properties** dialog.

4.1.3 Code Samples View

The File Properties dialog displays sample files that maybe be referenced during program development. The files shown are the contents of the `samples` directory contained in the selected toolsuite's distribution. These files may opened in an Editor view by either double-clicking their icon, or dragging the file icon to a Workspace view.

The files shown in this view are not stored in the project file.

Figure 4.2: File properties dialog



4.2 The Build Views

The views within the Build area display information relating to compilation of the project. The Build views are static views that are always visible and which are managed by HI-TIDE. There are four tabs, labelled: **Error Log**, **Memory Usage**, **Psect Usage** and **Build Log**. Each view is described in the following sections.

The terms *build* and *building* used in the following sections refer to either compiling, or compiling and linking of the files in the project. Figure 4.3 shows the Build area circled.

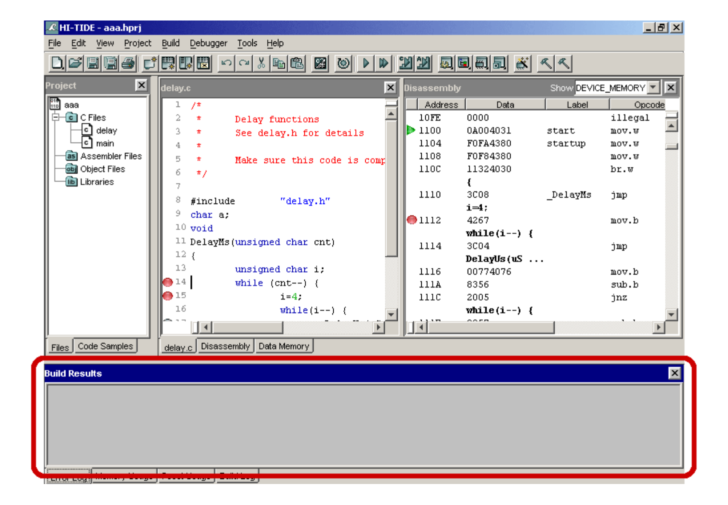
4.2.1 Error Log View

The Error Log view displays error and warning messages that were issued by the compiler when building the project. The errors/warnings are displayed in a table with four columns, marked: **Type**, **File**, **Line #** and **Description**.

Double clicking on a error or warning will display the file that contains the error and move the caret to the line on which the error occurred. Figure 4.4 shows a typical error summary in this tab, showing both warning and error messages.

The **Type** column displays easy-to-identify images which indicate either an error, warning or a successful build. A red **E** shown in the type column denotes an error occurred while building and the row that it appears on contains the error message that was issued by the compiler. A yellow **W**

Figure 4.3: Build area



in the type column denotes a warning message, issued by the compiler. If the build was successful, a green tick will be displayed.

The **File** column displays the file where the error or warning occurred. Some errors or warnings, such as linker errors, may not display a file name.

The **Line #** column indicates the line number of the file, shown in the **File** column, where the error or warning occurred. Some errors or warnings, such as linker errors, may not display a line number.

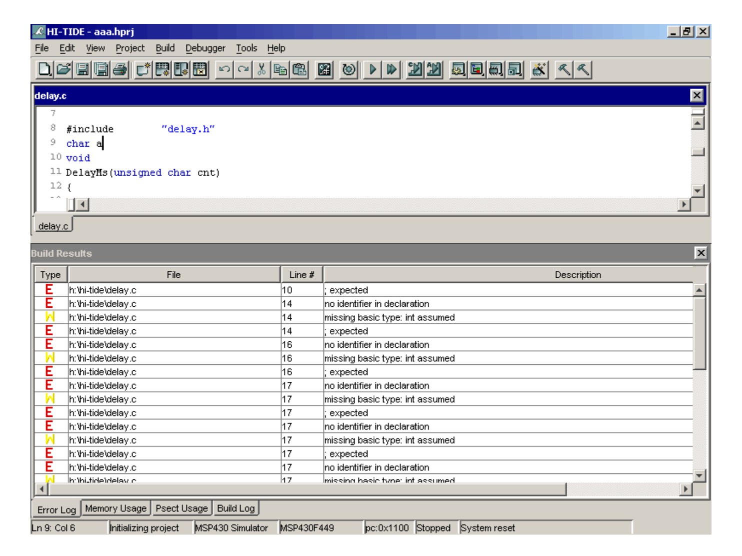
The **Description** column displays a short description of the error or warning that occurred. A successful build will yield the text `No Errors`. in this column.

4.2.2 Memory Usage View

The Memory Usage view displays memory usage statistics for the compiled program. This tab will only show the memory statistics if the build was successful, i.e. there were no errors. If errors occurred during in the build of the project, this tab will be blank.

The memory information that is displayed will be compiler specific. A typical memory usage output is shown in Figure 4.5.

Figure 4.4: Error log



4.2.3 Psect Usage View

The Psect Usage view displays psect information for the program sections in the compiled program. This tab will only show the psect usage if the build was successful , i.e. there were no errors. If errors occurred during the build of the project, this tab will be blank.

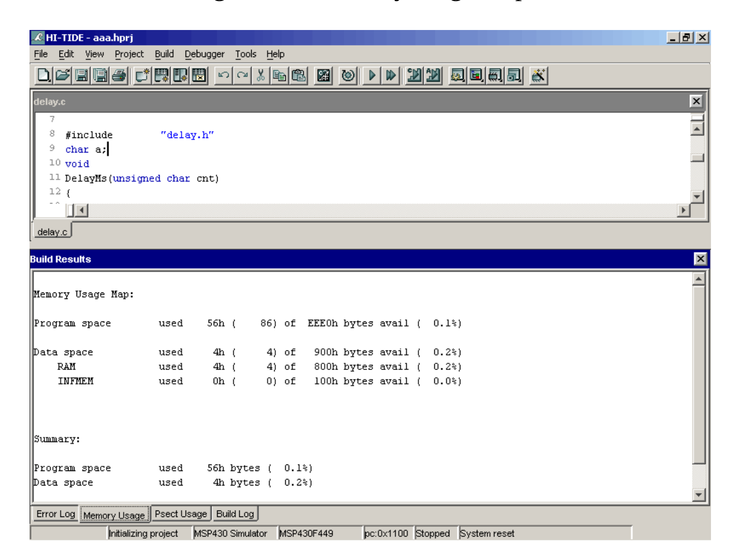
The psect usage information that is displayed will be compiler specific. Figure 4.6 shows a typical psect usage output.

4.2.4 Build Log View

The Build Log view displays detailed information on the build process. The information in the build log includes the date and time the build occurred, the dependency checking process used by HI-TIDE, command line options passed to the compiler to build and link the files and any compiler output.

The build log is updated on each build and will be updated even if errors occurred during building of the project. For a detailed description of when the build log is updated and the meaning of the contents of the log see Section 8.3.4.

Figure 4.5: Memory usage output



4.3 The Editor View

The Editor view is used to display and edit text files in HI-TIDE. The editor provides syntax highlighting for C source and header files. The editor also contains cut, copy and paste functionality and multi-level undo and redo.

The editor also provide a means of debugging source code, by allowing the setting of source-level breakpoints and tracing the code execution.

The Editor view consists of three regions - the line number gutter, the breakpoint gutter and the main text area, as shown in 4.7.

4.3.1 Editor Gutters

4.3.1.1 Breakpoint Gutter

The Breakpoint gutter, on the very left of the Editor view, is to provide a view of the breakpoints that have been set for the file showing in the Editor view. The items that can be shown in the breakpoint gutter are a “red dot”, a “grey dot”, a “green arrow” and a “red arrow”.

A “red dot” denotes that an enabled breakpoint has been set for the source code line on which the dot is shown. Breakpoints can be disabled without having to remove the breakpoint entirely. A “grey

Figure 4.6: Psect usage output

```
HI-TIDE - aaa.hprj
File Edit View Project Build Debugger Tools Help

delay.c
 7
 8 #include        "delay.h"
 9 char a;
10 void
11 DelayMs(unsigned char cnt)
12 {
```

```
Build Results
Psect Usage Map:

Psect     | Contents                          | Memory Range
----------|-----------------------------------|-------------------------------
program   | Startup code                      | 1100h - 1103h       4 bytes
init      | RAM clear and copy code           | 1104h - 110Fh      12 bytes
text      | Program and library code          | 1110h - 1135h      38 bytes
vectors   |                                   | FFE0h - FFFFh      32 bytes
----------|-----------------------------------|-------------------------------
bss       | RAM variables                     | 0200h - 0203h       4 bytes
----------|-----------------------------------|-------------------------------

Error Log | Memory Usage | Psect Usage | Build Log
Initializing project | MSP430 Simulator | MSP430F449 | pc:0x1100 | Stopped | System reset
```

Figure 4.7: Editor view layout

```
12 {
13         unsigned char i;
14         while (cnt--) {
15                 i=4;
16                 while(i--) {
17                         DelayUs(uS_CNT);
18                 } ;
19         };
20 }
21
```

dot” denotes that a breakpoint has been set at the source code line but the breakpoint is disabled. Source code lines that do not have any dots beside them means that there are not breakpoints set for that source line. Figure 4.7 shows breakpoints set, breakpoints enabled and disabled.

source-level breakpoints can be managed by right-clicking in the breakpoint gutter and selecting from popup menu or by double-clicking in the breakpoint gutter at the line of code. Refer to Section 4.3.8 for more details on the popup menu. Refer to Sections 4.3.9 and 4.3.11 for more details on managing source-level breakpoints from within the breakpoint gutter.

The breakpoint gutter also provides an indication of the program counter. While stepping the debugger, a program counter indicator (“green arrow”) will appear in the breakpoint gutter. This denotes the source statement that will be executed next. If the debugger stops at an assembler instruction that is within the block of instructions corresponding to a C statement, the arrow will point to that C statement. If the debugger stops on an activated source-level breakpoint, a “red arrow” will be displayed to show that the debugger has stopped on a breakpoint.

4.3.1.2 Line Number Gutter

The Line Number gutter is to the immediate right of the breakpoint gutter (see Figure 4.7). The line number gutter displays sequential line numbers with every line of source code. The line numbers form no part of the source file or program, but can be used to make reference to particular source lines easier.

4.3.2 Creating Editor Files

A new file can be created by selecting **New File** from the **File** menu or by clicking on the **New File** button in the standard toolbar. A new Editor view will be created and displayed in a new Workspace tab. A new file will be opened in the Editor view, in a tab called **Untitled*n***, where ***n*** is a number.

•

The new file will not be saved to disk until it is explicitly saved. Closing the file without saving will lose all unsaved data. When the file is saved, a file name and directory of the file can be set.

A new file can also be created from the project view by right clicking on a **C Files** or **Assembler Files** folder and selecting **Create and Add** menu. This will display an Editor view in a new Workspace tab and also save the file to disk.

Alternatively, new editor files can be created by dragging the **New File** button from the Standard toolbar to a workspace view. The mouse pointer changes to indicate that the view over which the pointer sits can be replaced by the editor.

4.3.3 Opening Editor Files

There are a number of different ways to open files in the Editor view.

A file can be open by selecting **Open File** from the **File** menu. This action will display a file dialog that will allow a file to be selected and displayed. Opening a file through this menu will create a new view in a new Workspace tab, labelled with the name of the file. The editor also stores a list of files that have been recently opened.

To open a file that has been recently opened, select the file from the **Open Recently Opened File** submenu in the **File** menu. This action will display the selected file in an Editor view in a new Workspace tab. The Workspace tab will be labelled with the name of the file. The number of files stored in the **Open** Recently Opened Files can be configured in the **General Preferences** dialog.

A C or assembler source file that is part of the project can be opened by double clicking on the file's icon in the Files view in the Project area. This action will display the selected file in an Editor view in a new Workspace tab if the file is not already opened. If the file is already opened in an editor, the editor will locate the first view that contains the file and give focus to that view.

A C or assembler file that is part of the project can also be opened by dragging the file from the project view and dropping the file onto an existing view. When the file is dropped, the existing view will be replaced by an Editor view displaying the selected file.

Files can also be reloaded if they are externally modified. By default the user is prompted to reload the file when it is detected as being modified externally, but this can be changed in the **General Preferences** dialog.

4.3.4 Saving Editor Files

To save a file, make sure the file is in focus and select **Save File** from the **File** menu. If the file is a new untitled file, a file dialog will be displayed where a file name and path for the file can be selected. If the file is not in focus, selecting the **Save File** menu item will have no effect.

Selecting **Save All** from the **File** menu will save the project file and all opened files in all Editor views.

To save a file under a different name and/or directory select **Save File As** from the **File** menu. When this action is selected, a file dialog will be displayed where a different file name and path for the file can be selected.

Files can also be configured to save automatically when building or when closing a project. By default, all files are saved when a project is about to be built, but this can be changed in the **General Preferences** dialog.

When a project is closing, HI-TIDE will, by default, prompt to save any modified files. This action can be changed in the **General Preferences** dialog.

4.3.5 Closing Editor Files

A file is considered closed when all of the Editor views displaying the file are closed. **Editor views** are closed in the usual way: selecting **Close View** from the **View** menu, or by clicking on the close button in the view's title bar to close the tab that contains the view.



4.3.6 Printing Editor Files

Afile open in an editor can be printed by focusing the Editor view and then selecting the **Print...** option from the **File** menu, or by clicking the **Print** button in the toolbar. The Print dialog will appear, to allow print options to be set. The appearance of the print dialog will be platform specific.

If more than one Editor view is opened in a Workspace tab, take care to ensure that the correct file is focused, otherwise the incorrect file may be printed.

The editor provides additional print options, such as printing line numbers and line wrapping, that might be useful when printing program code. See Section 2.3.1.2 for more details on these print options.

4.3.7 Syntax Highlighting

The Editor view uses a colour coding scheme to highlight the syntax of C files and header files. The editor detects if the file opened is a C file or a header file by the file's extension. C files have the extension `.c` and header files have the extension `.h`.

4.3.8 Editor Popup Menu

Right-clicking on any of the Editor views will display the Editor popup menu. The following describes the items in that menu that are specific to the Editor view. The view control menu items which will appear in the Editor's popup menu are described in Section 2.2.3.5.

Set Breakpoint Selecting this option sets a source-level breakpoint in the debugger currently selected in HI-TIDE. The breakpoint is set on the line over which the popup menu is raised. This option is only enabled if a debugger is selected in HI-TIDE, a HEX file is loaded and the source code actually defines executable assembler instructions. See also Section 4.3.9.

Remove Breakpoint This menu item replaces the **Set Breakpoint** menu item in the popup if the mouse is right-clicked over a source line that already has a breakpoint set at that location. This option removes the breakpoint entirely from the debugger.

Remove All Breakpoints Clears all of the breakpoints that have been set in the debugger.

Disable Breakpoint This turns off the breakpoint without removing the breakpoint. The breakpoint can be re-enabled by the **Enable Breakpoint** option.

Enable Breakpoint This menu item replaces the **Disable Breakpoint** menu item if the mouse is right-clicked over a source line that already has a disabled breakpoint set at that location. This option re-enables the disabled breakpoint.

Disable All Breakpoints This option deactivates all of the breakpoints that are currently set. This menu item is only enabled if there have been breakpoints set in the debugger. If there are no breakpoints set, this menu item will be disabled. This option does not affect disabled breakpoints.

Enable All Breakpoints This menu item activates all of the disabled breakpoints. This menu item is only enabled if there have been breakpoints set in the debugger. If there are no breakpoints, this option is disabled. This option does not affect enabled breakpoints.

Cut This menu item performs the editor cut selected text operation See 4.3.8.

Copy This menu item performs the editor copy selected text operation See 4.3.8.

Paste This menu item performs the editor paste text See Section 4.3.8.

4.3.9 Setting Source-Level Breakpoints

Source-level breakpoints can only be set on the line of source code, which generates an instruction or symbol. Not all C statements generate executable code. An example of such a statement is the declaration of an unitialized local variable. In some instances, the optimizer may remove or merge the generated executable instructions associated with source code which may result source code that does not referencing assembler instructions. Source-level breakpoints cannot be set for these lines.

To set a breakpoint, right-click the mouse over the line where the breakpoint is to be set. The Editor view popup menu will appear. If a valid source-level breakpoint is available for that line of source code, the **Set Breakpoint** option will be enabled in the popup menu. If a valid source-level breakpoint is not available, then the **Set Breakpoint** option will be disabled. If a breakpoint is already set at that line, the option **Remove Breakpoint** will appear in lieu of **Set Breakpoint**.

A source-level breakpoint can also be set by double-clicking on the breakpoint gutter or the line number gutter. If a valid source-level breakpoint is available for that line of code, then the breakpoint will be set. If a valid breakpoint is not available, double-clicking in the breakpoint gutter or line number gutter for that line will have no effect, that is, the breakpoint will not be set. Double-clicking on a breakpoint that is already set will activate or deactivate the breakpoint. See Section 4.3.11 for more details on activating and deactivating of breakpoints.

When a source-level breakpoint is set for a line of source code, a “red dot” will appear in the breakpoint gutter. This is shown in Figure 4.7, where lines 14 and 15 of the source code have enabled breakpoints set.

When a source-level breakpoint is set, a breakpoint will be set in the debugger at the assembly level at the assembly instruction or symbol that maps to the source code line. The assembly breakpoint will be shown in the Disassembly view. Refer to Section 4.4.1.4 for more details on breakpoints in the Disassembly view.

4.3.10 Removing source-level Breakpoints

To remove a source-level breakpoint, right-click over the breakpoint location and select the **Remove** Breakpoint option from the popup menu. The red dot or grey dot will be removed when the breakpoint is removed. Removing a source-level breakpoint will also remove the assembly level breakpoint that maps to the source-level breakpoint.

To remove all breakpoints, right-click in the Editor view and select **Remove All Breakpoints**. Breakpoints set in both the Editor view and Disassembly view will be removed. Refer to Section 4.4.1.5 for more details on removing breakpoints in the Disassembly view.

4.3.11 Activating/Deactivating source-level Breakpoints

Sometimes it is more desirable to disable a breakpoint than to remove the breakpoint altogether. This allows the user to temporarily deactivate that breakpoint, to later reactivate it, without having to remember where it was set.

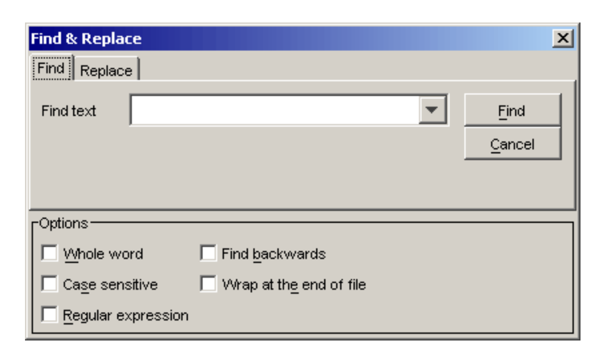
To disable an activated breakpoint, right-click over the source-code line at which the breakpoint is set and select **Disable Breakpoint** from the popup menu. The **Disable Breakpoint** menu item is only available if an activated breakpoint is set at that source code line location. See Section 4.3.8 for more details on the popup menu. Alternatively, double-clicking on the activated breakpoint (“red dot”) in the breakpoint gutter will deactivate the breakpoint. Disabling of the breakpoint will be denoted by the “grey dot”. Figure 4.7 shows a deactivated breakpoint at line 17 of the source code.

All breakpoints can be disabled by selecting the **Disable All Breakpoints** option from the Editor view’s popup menu.

Breakpoints can be enabled by right-clicking on the Editor view at the source-code line with the breakpoint and selecting **Enable Breakpoint** from the popup menu. The **Enable Breakpoint** menu item is only available when the selected line of source code has a disabled breakpoint set at that location. See Section 4.3.8 for more details on the popup menu. Similar to disabling breakpoints, a deactivated breakpoint can also be activated by double-clicking on it. The deactivated breakpoint (shown with a “grey dot”) will change to a “red dot”.

Deactivated breakpoints can also be activated by selecting the **Enable All Breakpoints** option from the Editor view popup menu.

Figure 4.8: Find and Replace dialog — find



4.3.12 Searching For Text

The editor can search for text and regular expressions, and replace text using the **Find & Replace** dialog. To open the Find & Replace dialog, select the **Find** menu item from the **Edit** menu. Figure 4.8 shows the dialog with the **Find** tab selected. Clicking on the **Find** button will perform a search of the current Editor view for the text shown in the **Find text** text field.

Figure 4.9 shows the dialog with the **Replace** function tab selected. Clicking on the **Find** button will search the current Editor view for the text shown in the **Find text** text field. Clicking on the **Replace** button will do a find and replace action. That is, any currently found text will be replaced prior to the dialog locating the next occurrence of the text in the **Find text** text field.

The **Find text** text field accepts regular expressions as part of the search string. A table of the accepted regular expressions and their meaning is tabulated in Section D.

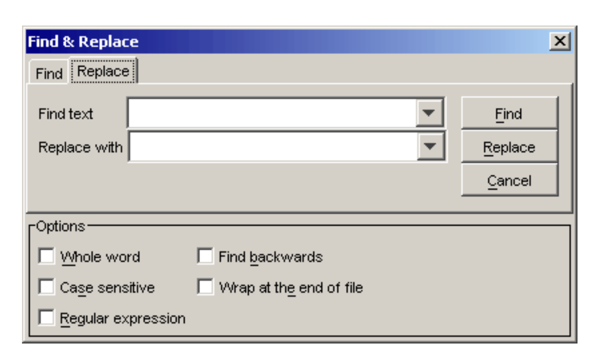
4.3.13 Search Options

By default, searches are performed from the current location of the editor caret, left-to right and downwards in the editor document. The searched text can be part of a word or a whole word and will be case-insensitive. The search will stop at the bottom of the document.

Options in the **Find & Replace** dialog allows refinement of the search. As the options are checkboxes, any combination of the options can be selected. Each option is described individually in the following sections.

Whole word Selecting this option will restrict the text found to those that are whole words only. For example, if the search text is `in`, the possible matches will be: `in`, `In`, `iN` or `IN`. If this option

Figure 4.9: Find and Replace dialog — replace



is unchecked, the search text may also form part of a larger word, such as; `int`, `include` or `BIN`. Selecting the **Whole word** search will disable the search for **Regular expression** option.

Match case When this option is selected the search will only find words or expressions that are of the same case as the search text. For example, searching for `start` will not find `Start` or `START`. Unchecking this option will make the search case-insensitive and would, for the same example search string, match `start`, `Start`, `sTaRt` etc.

Regular expression Selecting this option makes the text in the **Find text** text field a regular expression rather than a text string. Selecting the **Regular expression** search option will disable the search for **Whole word** option.

Find backwards By default, the search is performed from the current caret position, going left to right and downwards in the editor document. If a caret was to the right or below a string or expression being searched for, it will not be located. Selecting the **Find backwards** option changes the search to start from the current caret position, going right-to-left and upwards in the editor document. Deselecting this option will return it to the default search order.

Wrap at the end of file By default, if searching in default order, the search will stop once the end of the file is reached. If the **Find backwards** option is selected, the search will stop at the top of the file. Selecting the Wrap at the end of file option will allow the search to "wrap" around the document. That is, if the search is in the default order, once the search reaches the end of the file, it will start searching from the top of the file again. If the **Find backwards** option is selected, the search will restart from the bottom of the file once it has reached the top of the file.

4.4 The Debugger Views

Several of the available views are collectively known as the Debugger views. These views interact with the selected debugger. If no debugger is selected then these views are not applicable and cannot be displayed. These views include the Disassembly view, Data Memory view, Registers view, and Variable Variable Watch view, Local Variable Watch view.

If any of these views are displayed and the debugger is changed to no debugger, then the view will remain in the Workspace area, but become blank, i.e. have no contents, although the title bar and layout will remain. Blank views may be closed in the usual way, and if a debugger is again selected, the view display will be updated.

4.4.1 Disassembly View

The Disassembly view provides a view of the executable memory of the target device, as well as a means to step through the code and set breakpoints in the code. The following sections describe the Disassembly view in detail.

4.4.1.1 Disassembly View Layout

The **Disassembly view** consists of a tabulated view, with 6 columns (see 4.10). Each row of the view is shown as a disassembled assembler instruction. The Disassembly view always interprets the memory as executable assembler instructions, even if the bytes located here are program data.

The first column on the left of the view is the Breakpoint gutter. This is similar to the Breakpoint gutter in the Editor view. The Breakpoint gutter allows setting and viewing of breakpoints, as well as tracing the program counter. See Section 4.4.1.2 for more details.

The second column from the left side of the view, labelled **Address**, displays the starting address of the program memory being displayed. As instructions may be of fixed width, the addresses may not increase linearly.

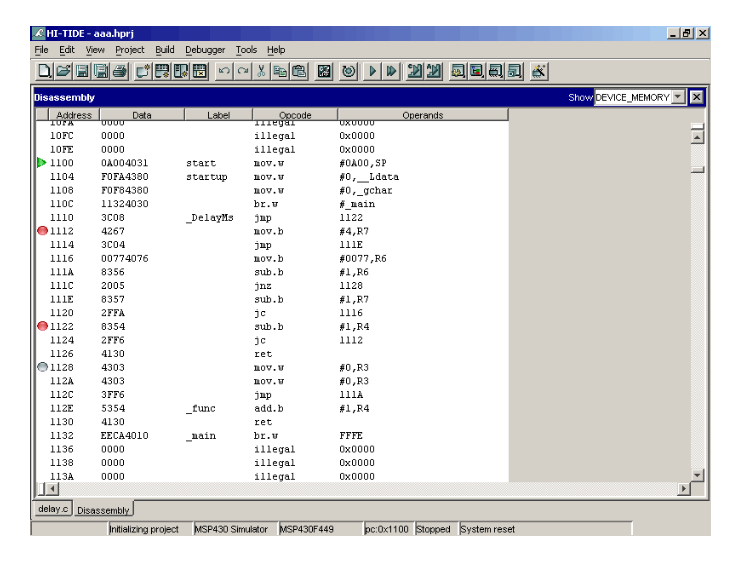
The **Data** column shows the target device's numeric machine code corresponding to the instruction represented by the line. The codes shown will vary depending on the instruction set of the target device.

The **Label** column displays any symbol that is associated with the address of the assembled line. Note that there can be more than one label at the same address. The label that is last read from the debugging file will be the one displayed. If there is no label available for the assembled line, the entry in the **Label** column will be left blank on this line.

The **OpCode** column shows the human-readable interpretation of the machine code instruction.

The **Operands** column shows the operands used with the opcode. These are usually displayed as addresses in hexadecimal format. If a label or register name is found at the referenced address, the human-readable form of that address is displayed.

Figure 4.10: Assembler view



The Disassembly view title bar contains a combo box which contains the names of memory spaces available on the device that is selected. Harvard architecture devices may contain more than one memory space. The disassembled memory shown in this view will be that from the selected memory space.

4.4.1.2 Breakpoint Gutter

The breakpoint gutter in the Disassembly view shows the assembly level breakpoints and also traces the program counter. It also allows the setting, deactivating and activating of breakpoints. The items that are shown in the breakpoint gutter are a “red dot”, a “grey dot”, a “green arrow” and a “red arrow”. See 4.10.

A “red dot” denotes that an *activated* breakpoint has been set for the memory address represented by the assembled line of program memory. A “grey dot” denotes that a breakpoint has been set at that address but the breakpoint is disabled.

Assembly level breakpoints can be managed by right-clicking in the breakpoint gutter and selecting from the popup menu or by double-clicking in the breakpoint gutter at the assembly line.

The breakpoint gutter also provides a trace of the program counter. While stepping the debugger, a program counter indicator (“green arrow”) will appear in the breakpoint gutter. This denotes an

instruction that is to be executed next. If the debugger stops on an activated breakpoint, a "red arrow" will be displayed to show that the debugger has stopped on a breakpoint.

4.4.1.3 Disassembly View Popup Menu

Right-clicking on any of the Disassembly views will display the Disassembly view popup menu. The following describes the items in that menu that are specific to the Disassembly view. The view control menu items are described in Section 2.2.3.5.

The Disassembly view can have its colours and font customised. This is set via the **Font/colours...** popup menu item. The **Font/colour Settings** dialog is described in Section 2.2.3.6.

Set Breakpoint Selecting this option sets an assembly-level breakpoint in the current debugger. This option is only enabled if a debugger is selected in HI-TIDE and a HEX file has been loaded.

Remove Breakpoint This menu item only appears (in lieu of **Set Breakpoint**) if the mouse is right-clicked over assembled line that already has a breakpoint set at that location. When **Remove** Breakpoint is displayed, the **Set Breakpoint** menu item will not be displayed.

Remove All Breakpoints Clears all of the breakpoints that have been set in the debugger.

Disable Breakpoint This turns off the breakpoint without removing the breakpoint. The breakpoint can be enabled by the **Enable Breakpoint** option.

Enable Breakpoint This menu item only appears (in lieu of **Disable Breakpoint**) if the mouse is right-clicked over Disassembly view line that already has a breakpoint set at that location and that breakpoint is disabled itself.

Disable All Breakpoints This option deactivates all of the breakpoints that are currently set. This menu item is only enabled if there have been breakpoints set in the debugger. If there are not breakpoints set, this menu item will be disabled. Selecting this option on breakpoints that have been deactivated will have no effect.

Enable All Breakpoints This menu item activates all of the breakpoints that are currently set. This menu item is only enabled if there have been breakpoints set in the debugger. If there are not breakpoints, this option is disabled. Selecting this option on breakpoints that are activated will have no effect.

Show PC Selecting this option displays the program memory, in the Disassembly view, starting from the address of where the program counter is at. See also Section 4.4.1.7.

Figure 4.11: Breakpoints in assembler view

Disassembly

Address	Data	Label	Opcode	Operands
10FE	0000		illegal	0x0000
1100	0A004031	start	mov.w	#0A00,SP
1104	F0FA4380	startup	mov.w	#0,__Ldata
1108	F0F84380		mov.w	#0,_gchar
110C	11324030		br.w	#_main
1110	3C08	_DelayMs	jmp	1122
1112	4267		mov.b	#4,R7
1114	3C04		jmp	111E
1116	00774076		mov.b	#0077,R6
111A	8356		sub.b	#1,R6
111C	2005		jnz	1128

Show C Source Selecting this option displays the C source code line that was compiled to produce the compiler instructions displayed. Deselecting this option hides the C source code. Figure 4.10 shows an example of an Disassembly view with C source code showing. Figure 4.11 shows an example of an Disassembly view without C source code showing. Refer to Section 4.4.1.8 for more details.

Track PC Location The option is a check box option. When the option is checked, the Disassembly view will automatically update the view to follow the location of the program counter. Unchecking the option will turn off this feature. See also Section 4.4.1.7.

4.4.1.4 Setting Assembly Level Breakpoints

Unlike source-level breakpoints, assembly level breakpoints can be set at any assembly line, since each assembly line is executable. When a source-level breakpoint is set, the breakpoint will also appear in the Disassembly view. Setting an assembly level breakpoint will not necessarily make the breakpoint appear in the Editor view, as not all source code will line up with assembly instructions.

To set a breakpoint, right-click the mouse over the line that the breakpoint is to be set. The Disassembly view popup menu will appear. If a breakpoint has not been set at that assembly line address, the **Set** Breakpoint option will be displayed in the popup menu. If a breakpoint is already set at that line, the option **Remove Breakpoint** will appear in lieu of **Set Breakpoint**.

A source-level breakpoint can also be set by double-clicking on the breakpoint gutter next to the address of where the breakpoint is to be set. When the new breakpoint is set, a “red dot” will appear in the breakpoint gutter (as shown by the line at Address 1110 in Figure 4.11).

Double-clicking on a set breakpoint will either activate or deactivate that breakpoint. If a breakpoint is set and activated (red), double-clicking on it will deactivate that breakpoint (as shown by

the line at Address 1116 in 4.11). See Section 4.4.1.6 for more details on activating and deactivating breakpoints. Double-clicking on a deactivated breakpoint will activate that breakpoint.

4.4.1.5 Removing Assembly Level Breakpoints

To remove an assembly level breakpoint, right-click over the set breakpoint and select the **Remove** Breakpoint option from the popup menu. The red or grey dot denoting the breakpoint will be removed. If the assembly line mapped to a line of source code, the breakpoint will also be removed from the Editor view.

To remove all breakpoints, right-click in the Disassembly view and select **Remove All Breakpoints**. Breakpoints set in both the Disassembly view and Editor view will be removed. Refer to Section 4.3.10 for more details on removing breakpoints from the Editor view.

4.4.1.6 Activating/Deactivating Assembly Level Breakpoints

To deactivate an enabled breakpoint, right click on the assembly line that the breakpoint is set for and select **Disable Breakpoint** from the popup menu. The **Disable Breakpoint** menu item is only available if an activated breakpoint is set at that source code line location. See Section 56 for more details on the popup menu. Alternatively, double-clicking on the activated breakpoint ("red dot") in the breakpoint gutter will deactivate the breakpoint. Disabling of the breakpoint will be denoted by the "grey dot".

Breakpoints can be disabled, as a whole, by selecting the **Disable All Breakpoints** option from the Disassembly view popup menu.

Breakpoints can be enabled by right-clicking on the Disassembly view at the assembly line with the breakpoint and selecting **Enable Breakpoint** from the popup menu. The **Enable Breakpoint** menu item is only available when the selected assembly line has a disabled breakpoint set at that location. Similar to disabling breakpoints, a deactivated breakpoint can also be activated by double-clicking on it. The deactivated breakpoint (shown as a "grey dot") will change to a "red dot".

Deactivated breakpoints can also be activated by selecting the **Enable All Breakpoints** option from the Disassembly view popup menu.

4.4.1.7 Displaying Program Counter Location

The Disassembly view is capable of indicating where the program counter is at. While stepping the debugger, the value of the program counter is indicated by a "green arrow" (as shown by the line at Address 1100 in 4.11). This denotes the instruction that is to be executed next.

When the program counter changes, the "green arrow" will move to reflect the change and show the new location. The "green arrow" is only updated if the debugger is stepping - single stepping, continuous (*animating*) or source-level stepping (*C stepping*). If the debugger is running, the "green arrow" will not be updated until the debugger stops.

While the debugger is stepping, it is possible that the program counter may be at a value that is not currently displayed in the Disassembly view. There are two ways to display the assembly line where the program counter is at, if it is out of view.

Right-clicking in the Disassembly view and selecting the **Show From PC Value** option from the Disassembly view popup menu will display memory in the Disassembly view, with the first assembly line starting from the address of where the program counter is at.

Alternatively, the Disassembly view is able to automatically track the program counter so that if it is at a location that is not currently displayed, it will scroll to the location of where the program counter is at. This is similar to automatically selecting the **Show From PC Value** each time the debugger steps and the program counter is not displayed. To enable this feature, right-click on the Disassembly view and select the **Track PC Location** option. When enabled, the option will have a checked tickbox displayed next to the option. When disabled, the option will have an unchecked tickbox displayed next to it. See Section 4.4.1.3 for more details on the Disassembly view popup menu.

4.4.1.8 Displaying C Source Code

The Disassembly view is capable of displaying mixed C code and assembly code within the view. Each C code line shown is grouped with the block of assembler code that was generated from that C line. Thus, making the view similar to that of an assembler listing file. The number of assembler instructions that follow a C line will vary, depending on the C source code itself.

To display C source code in the Disassembly view, right-click on the Disassembly view and select **Show C** Source from the popup menu. The C code is shown in bold type and is placed in the Data column. 4.12 shows a typical Disassembly view with C code showing.

4.4.2 Data Memory View

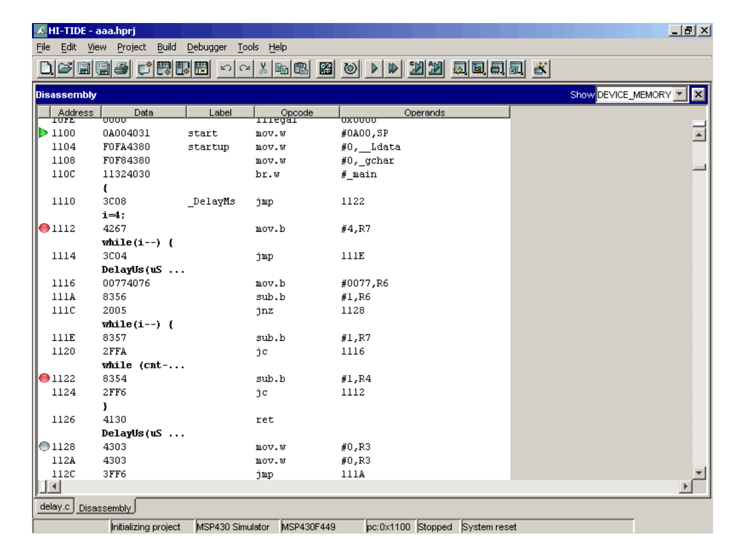
The Data Memory view displays the writable memory of the target device. It also shows the memory locations that have had their values changed, to aid in debugging. The view is described in detail in the following sections.

4.4.2.1 Data Memory View Layout

The Data Memory view displays the writable memory in a table format. Each row of the view displays a range of memory addresses and the ASCII value of each memory address. The number of addresses displayed in the row varies, depending on the width of the view, as well as the format of the data to be displayed. Figure 4.13 shows a typical Data Memory view.

On the very left of the Data Memory view, the **Address** column displays the starting address for each row. The addresses are always displayed in hexadecimal. On the far right, the Data Memory

Figure 4.12: Source code in assembly view



view always has an **ASCII** column. The **ASCII** column displays the ASCII value of each of the memory addresses for that row. The other columns display the contents of each of the memory locations in the row.

The number of memory columns shown per row will depend on the width of the format of the displayed data, however the number of bytes shown will always be 16.

The Data Memory view title bar contains a combo box which contains the names of memory spaces available on the device that is selected. Harvard architecture devices may contain more than one memory space. The displayed data memory shown in this view will be that from the selected memory space.

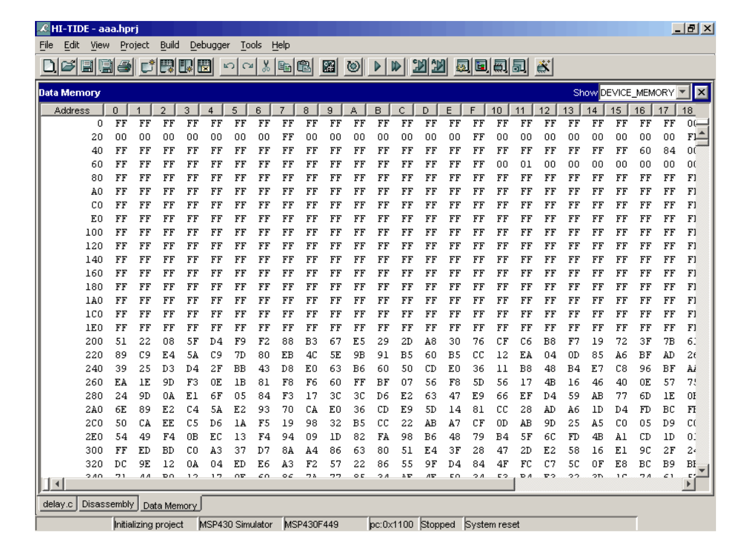
4.4.2.2 Data Memory View Popup Menu

Right-clicking on any of the Data Memory views will display the Data Memory view popup menu. The following describes the menu items that are specific to the Data Memory view. The view control menu items are described in Section 2.2.3.5.

The following three menu options change the radix of the displayed memory.

Hex Selecting this option displays all memory locations in hexadecimal.

Figure 4.13: Data memory view



Decimal Selecting this option displays all memory locations in decimal.

Octal Selecting this option displays all memory locations in octal.

The following three options changes the number of bytes displayed per location column in the Data Memory view.

Byte Selecting this option displays one byte of memory per location column in the view.

Word Selecting this option displays the number of memory locations required to show a word per location column. The number of bytes per word will depend on the device selected.

Long Selecting this option displays the number of memory locations required to show a long type, per location column. The number of bytes per long will depend on the selected device.

4.4.2.3 Tracing Memory Usage

Each time the debugger is stopped, the Data Memory view updates the values displayed. The memory locations whose contents have changed since the debugger was started (stepping, animating or running) will be highlighted in red. The locations whose contents have not changed are shown in

the normal font colour set for the view (black by default). Note: to improve performance, only those memory locations that have been previously displayed in the view are highlighted when changed and no updates are performed while the debugger is actually running.

4.4.2.4 Modifying Memory

The memory locations in the Data Memory view can be modified by the user. To modify the value stored at a memory location, click (or double-click) on that location and type the new value into the memory location cell. Pressing *enter* or clicking the mouse in any other column will modify the location. Pressing *escape* will cancel the change. Changed values will appear highlighted in red type.

To prevent accidental changes in this view, click the mouse on any column other than the memory columns to deselect any selected memory location cell.

The new value must be specified as a hexadecimal number. If the new value is not a valid hexadecimal number, no change is made. If the new value is a valid hexadecimal number, but is too large for the the memory location, it will be truncated to fit. For example, if an attempt is made to change a 16-bit wide word location to the hexadecimal value `123456`, the location will be assigned the value `3456`.

4.4.3 Registers View

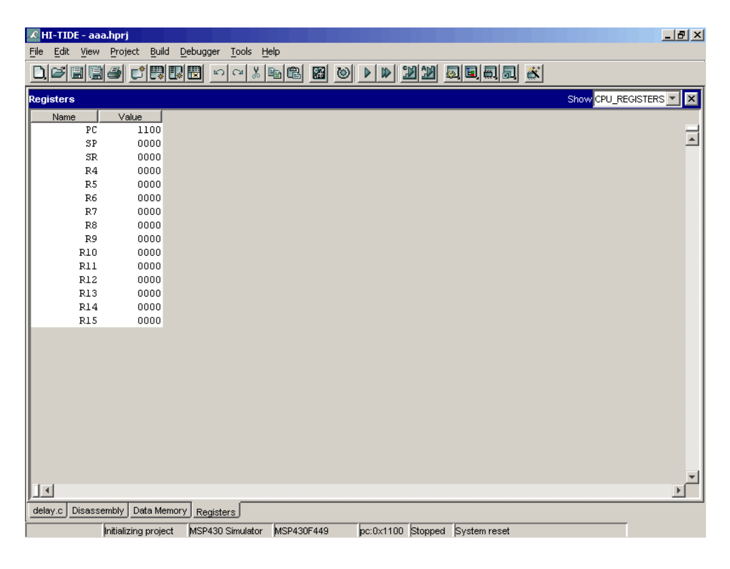
The Registers view displays the registers of the target device. The view is described in detail in the following sections.

4.4.3.1 Registers View Layout

The Registers view contains two columns: **Name** and **Value**. The **Name** column displays a human readable form of the register's name, while the **Value** column displays the contents of the register. The register's contents can be displayed in several radices which are selectable from the view's popup menu. Figure 4.14 shows a typical Registers view.

The Registers view title bar contains a combo box which contains the names of different types of registers available on the selected device. Typically a device will have CPU registers, e.g. accumulators and status registers; and special function registers — those registers used to control and monitor on-board peripherals. The registers shown in this view will be those specified by the combo box selection.

Figure 4.14: Registers view



4.4.3.2 Registers View Popup Menu

Right-clicking on any of the Registers views will display the Registers view popup menu. The following describes the menu items that are specific to the Registers view. The view control menu items are described in Section 2.2.3.5.

The Registers view can have its colours and font customised. This is set via the **Font/colours...** popup menu item. The **Font/colour Settings** dialog is described in Section 2.2.3.6.

The following three menu options changes the radix of the displayed memory.

Hex Selecting this option displays all the memory values in hexadecimal.

Decimal Selecting this option displays the memory values in decimal

Octal Selecting this option displays the memory values in octal.

Binary Selecting this option displays the memory values in binary.

4.4.3.3 Tracing Register Usage

Each time the debugger is stopped, the Registers view updates the values displayed. The registers whose contents have changed since the debugger was started (stepping, animating or running) will

be highlighted in red. The registers whose contents have not changed are shown in the normal font colour set for the view (black by default). Note: to improve performance, only those memory locations that have been previously displayed in the view are highlighted when changed. Note: to improve performance, only those memory locations that have been previously displayed in the view are highlighted when changed and no updates are performed while the debugger is actually running.

4.4.3.4 Modifying Memory

The registers in the Registers view can be modified by the user. To modify the value stored in a register, click (or double-click) on the desired register's **Value** column cell and type the new value. Pressing *enter* or clicking the mouse in any other column will modify the location. Pressing *escape* will cancel the change. Changed values will appear highlighted in red type.

•

To prevent accidental changes in this view, click the mouse on any column other than the **Value** column to deselect any selected register contents cell.

The new value must be specified as a hexadecimal number. If the new value is not a valid hexadecimal number, no change is made. If the new value is a valid hexadecimal number, but is too large for the the register, it will be truncated to fit. For example, if an attempt is made to change a 16-bit wide register to the hexadecimal value `123456`, the register will be assigned the value `3456`.

4.4.4 Variable Watch View

The Variable Watch view is a view for monitoring non-local program variables. Specifically it can display all variables that are not defined within a C function. The variables are represented in the symbolic debug information file, or SDB file. SDB files are generated by the code generator and one is produced for each C source file in the project.

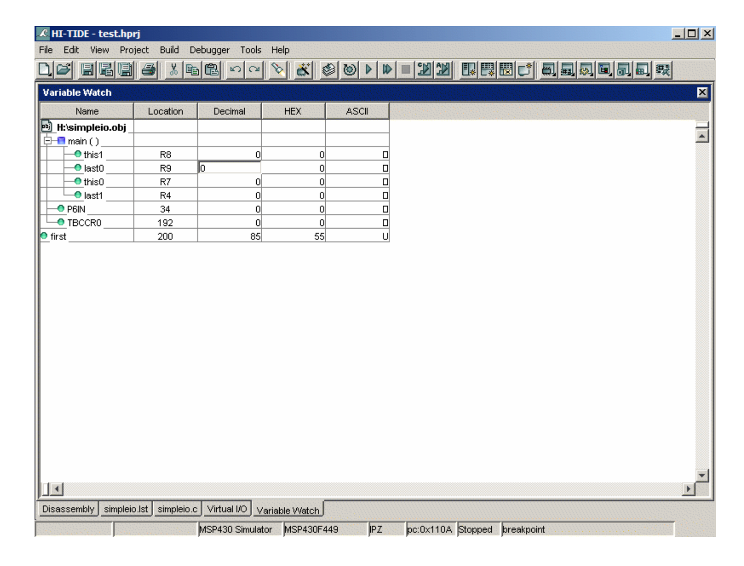
The view is described in detail in the following sections.

4.4.4.1 Variable Watch View Layout

The Variable Watch view is composed of several columns. Each column has a label at the top of the view. The **Name** column is always present and by default a **Location**, **Decimal**, **HEX** and **ASCII** contents column are also displayed. Additional columns can be added via the Variable Watch popup menu. Figure 4.15 shows a typical Variable Watch view.

The width of the columns can be adjusted. As the mouse pointer is moved over the divider of the column name, it changes to a horizontal resize cursor. Click and drag the divider to the required position. All the available columns in the Variable Watch view are summarized below.

Figure 4.15: Variable Watch view



Name This permanent field shows the name of the C identifier being displayed. This is not the symbol that would be used in assembler code to access this variable. An icon is used to show the type of the variable represented by the symbol and a tree structure is used to indicate the scope of the variable within the program's hierarchy. The icons and symbols are fully described in Section 4.4.4.2.

Location displays the location of the variable. The location can either be the hexadecimal address of the memory that holds the specified memory or a register name.

Type displays the C types of displayed variables.

Decimal/HEX/ASCII/Binary displays the contents of the variable in decimal, hexadecimal, ASCII character and binary format, respectively. The string `Out of scope` is displayed if the variable is not legally accessible at the point at which the program is stopped.

4.4.4.2 Variable Icons and Tree Representation

The Name column in the Variable Watch view uses icons and a tree structure is used to indicate the scope of the variable within the program's hierarchy. This view can be used to display the contents

of both local and global objects so some means of indicating the scope of the variables is required since there may be more than one variable with the same name.

If any variables added to this view are local to a function or block of code within a function, a bolded row which contains an object file icon and the object filename is displayed. In a tree emanating down from this module name is a row which contains a square blue box and name of the function in which the variable is defined. Parentheses, `()`, are placed after the function's name. The local variables are shown in a subtree emanating down from this function name. Each variable has a green dot next to its name.

Variables which are defined outside a function, but which are `static` have scope only within the module in which they are defined. Such variables are shown in the tree emanating from the module name icon.

Variables defined outside a function and which have external linkage are shown independent of any tree and as the last rows in the view.

Objects of aggregate type (arrays and structures) can expanded to display the individual elements or members within the object. Array object names are followed by square brackets, `[]`, and structures are followed by braces, `{}`. Double-clicking the name column associated with any other these types will expand or collapse the contents of the type. The elements of an array are represented by their numerical index; structure members are represented by their member name. If the structure contains bitfield variables, a colon and the size of the bitfield is given following its name.

Pointer types can also be shown in an expanded state to reveal the contents of the variable or object to which it refers. They are initially shown expanded and a red arrow is displayed on the second line. The contents of the object to which the pointer refers is displayed on this line as well as its location in memory if the appropriate columns have been set up.

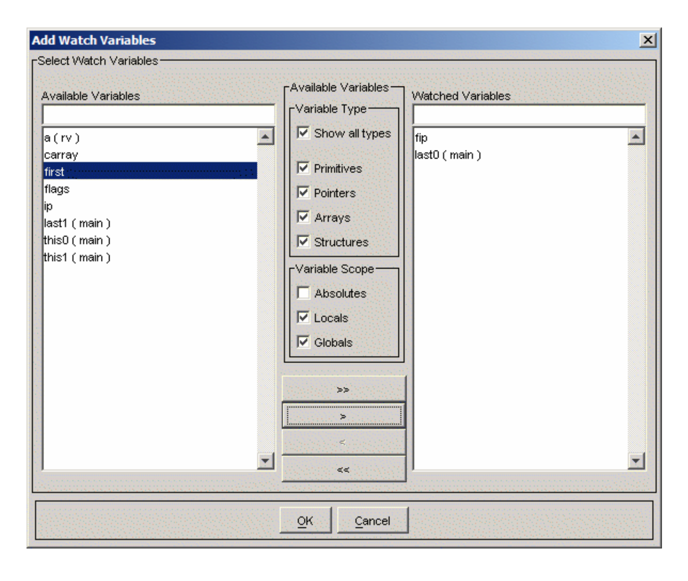
4.4.4.3 Variable Watch View Popup Menu

Right-clicking anywhere in the Variable Watch view window will display a popup menu. The following describes the menu items contained in this menu. The view control menu items are described in Section 2.2.3.5.

Add/Remove Variables Selecting this option will bring up a **Add Watch Variables** dialog as described in Section 4.4.4.4. Variables can be added to, or removed from, this view using this dialog.

Show *column* There is one menu item for each column that can be displayed in the Watch view, except the **Name** column which cannot be hidden from view, which can be used to show or hide the column. When a column is visible, a tick is shown next to the menu item.

Figure 4.16: Add/remove variables dialog



4.4.4.4 Adding and Removing Variables

Source variables can be added to, or removed from, the Variable Watch view at any time via the **Add Watch Variables** dialog as shown in Figure 4.16. This dialog is opened by selecting **Add/Remove Variables** menu item from the Variable Watch view popup menu.

On the left of this dialog is a scrollable text box which lists all variables that can be displayed in this view. This list is searchable. To find a known variable quickly, start typing the name of the variable into the text field above the list box. The first variable whose name matches the search string is selected in the list box. Continue typing letters of the variable's name until the required variable is selected. The search string is not case sensitive.

A variable is added to the view by selecting it and clicking the **Add selection** button. The component will then appear in the scrollable list box on the right of the dialog, under the **Watched Variables** label.

More than one variable may be added in one operation. Select all the variables required whilst holding down the *shift* key (consecutive selection) or *control* key (nonconsecutive selection), and then click the **Add selection** button. All the available components may be added simultaneously by clicking the **Add all** button.

As a program may contain a large number of variables, the number of variables shown in the

Available Variables list can be limited to variables of a particular type. Checkboxes are present in the centre of the dialog and have the following meanings.

Show all types Enables all the checkboxes under the **Variable Type** group.

Primitives Deselecting this option will remove primitive types (basic types such as `char`, `int` etc) from the list of available types which can be selected and displayed in the view. Conversely, checking this option will show and allow selection of these variables.

Pointers Deselecting this option will remove all pointer types from the list of available types which can be selected and displayed in the view. Conversely, checking this option will show and allow selection of these variables.

Arrays Deselecting this option will remove all array types from the list of available types which can be selected and displayed in the view. Conversely, checking this option will show and allow selection of these variables.

Structures Deselecting this option will remove all structure and union types from the list of available types which can be selected and displayed in the view. Conversely, checking this option will show and allow selection of these variables.

Absolutes Deselecting this option will remove all variables, regardless of their type, which are defined as absolutes from the list of available types which can be selected and displayed in the view. Conversely, checking this option will show and allow selection of these variables.

Locals Deselecting this option will remove all variables, regardless of their type, which are local (defined within a function) from the list of available types which can be selected and displayed in the view. Conversely, checking this option will show and allow selection of these variables.

Globals Deselecting this option will remove all variables, regardless of their type, which are global (defined outside a function) from the list of available types which can be selected and displayed in the view. Conversely, checking this option will show and allow selection of these variables.

Variables may be removed by opening the **Add Watch Variables** dialog by clicking the **Add/Remove Variables** menu item from the Variable Watch view popup menu. The variables already displayed in the view will be shown in the scrollable list box under the **Watched Variables** label. This list is searchable. To find a connected variable quickly, start typing the name of the variable into the text field above the list box. The first variable whose name matches the search string is selected in the list box. Continue typing letters of the variable's name until the required variable is selected. The search string is not case sensitive.

A variable is removed by selecting it and clicking the **Remove selection** button. The variable will then disappear from the list box.

More than one variable may be removed in one operation. Select all the variables required whilst holding down the *shift* key(consecutive selection) or *control* key (nonconsecutive selection), and then click the **Remove selection** button. All the available variables may be removed simultaneously by clicking the **Remove all** button.

4.4.4.5 Modifying Variables

The variables in the Variable Watch view can be modified by the user. To modify the value stored in a variable, click (or double-click) on any of the desired variable's contents columns (Decimal, HEX, ASCII, binary) and type the new value. Pressing *enter* or clicking the mouse in any other column will modify the location. Pressing *escape* will cancel the change. Changed values will appear highlighted in red type.

To prevent accidental changes in this view, click the mouse on any column other than the variable's contents columns to deselect any selected variable contents cell.

The new value must be specified in the same radix as the selected contents cell displays, e.g. if you are changing a **Decimal** display cell, then the new value is assumed to be decimal. If the new value is not valid, no change is made. If the new value is valid, but is too large for the the register, it will be truncated to fit. For example, if an attempt is made to change a 16-bit wide variable to the hexadecimal value `123456`, the variable will be assigned the value `3456`.

4.4.5 Local Watch View

The Local Watch view is a view for monitoring local program variables. Specifically it displays all variables that are defined within a C function, i.e. `auto` and `static` local objects whose scope is limited to a function or a block within a function. As these variables cannot be accessed when they are out of scope, this view automatically updates its contents during program execution with those local variables currently in scope. Variables cannot be manually added to, or removed from this view.

The operation of this view is identical to the Variable Watch view which is described in Section 4.4.4, with the exception of information relating to adding and removing variables from the view.

The name of the function in which scope is limited to is displayed in the title bar for this view.

4.4.6 Virtual I/O View

4.4.6.1 Overview

The Virtual I/O view is like an electronic test area where virtual components can be placed and wired to the microcontroller being simulated. A range of components is available which allow the operation of the program to be seen as well as allow the user to interact with the program. For example, an LED and push button switch might be wired to the peripheral port of the microcontroller. As a program is simulated, the LED will turn on and off as values are written to the port, and the switch may be clicked with the mouse which in turn changes values read back from the port. This might be used to verify that the settings associated with the port, such as the data direction register etc, are correctly configured.

The Virtual I/O view is primarily intended to be used when the simulator is selected as the debugger, in fact the simulator is considered as one of the available components that can be wired and represent the simulated microcontroller. However it is possible to use the Virtual IO without a simulator connected.

•

To quickly see the operation of the Virtual IO view without having to write or compile a program, add a push button and an LED and wire them together. As you push the button, you should see the LED illuminate.

The Virtual I/O view is initially empty when first opened. A grid is drawn over the view which can be used to help align components on the screen. The items in the Virtual I/O popup menu are described in the following sections.

4.4.6.2 Virtual I/O View Popup Menu

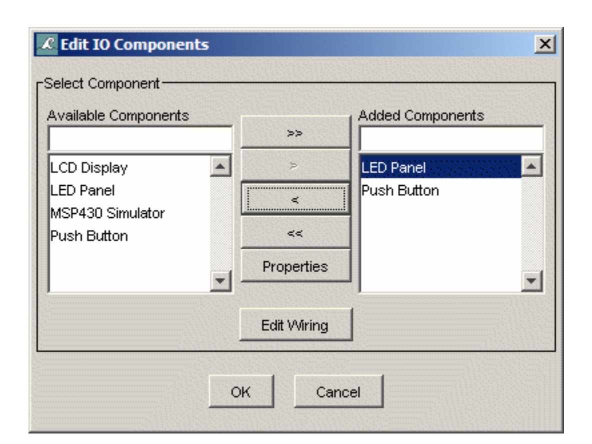
The view-control menu items for manipulating Workspace views are described in Section 2.2.3.5.

The following specific menu items are contained in the Virtual I/O view popup menu.

Add/Remove/Edit Component. . . Clicking this item opens the **Edit IO Components** dialog showing the **Select Component** display. This is shown in Figure 4.17. The full details of how to add and configure components are given in Section 4.4.6.3.

Edit Wiring. . . Clicking this item opens the **Edit IO Components** dialog showing the **Wire Component** display. This is shown in Figure . This menu item is only0 accessible after components have been added to the Virtual IO view. The full details of how to wire components are given in Section 4.4.6.6.

Figure 4.17: Edit IO Components dialog — Select component



4.4.6.3 Adding Components

Components can be added by opening the **Edit IO Components** dialog from the Virtual IO view popup menu. This dialog is illustrated in Figure 4.17.

A list of the all the available components is shown in the scrollable list box on the left side of the dialog, under the **Available Component** label. This list is searchable. To find a known component quickly, start typing the name of the component into the text field above the list box. The first component whose name matches the search string is selected in the list box. Continue typing letters of the component's name until the required component is selected. The search string is not case sensitive.

A component is added by selecting it and clicking the **Add selection** button. The component will then appear in the scrollable list box on the right of the dialog, under the **Added Components** label. A component may be added more than once by clicking the Add Selection button as many times as required.

More than one component may be added in one operation. Select all the components required whilst holding down the *shift* key (consecutive selection) or *control* key (nonconsecutive selection), and then click the **Add selection** button. All the available components may be added simultaneously by clicking the **Add all** button.

To complete the addition of the components, click **OK** to close the dialog. The new components will be shown in the Virtual IO view. Alternatively click **Edit Wiring** to save any components added and go to the next display in the **Edit IO Components** dialog. Wiring is discussed in Section 4.4.6.6.

4.4.6.4 Removing Component

Components may be removed from the Virtual IO view several ways. Components may be removed by selecting the **Add/Remove/Edit Component...** menu item from the Virtual IO view popup menu. The components already added will be shown in the scrollable list box under the **Added Components** label. This list is searchable. To find a connected component quickly, start typing the name of the component into the text field above the list box. The first component whose name matches the search string is selected in the list box. Continue typing letters of the component's name until the required component is selected. The search string is not case sensitive.

A component is removed by selecting it and clicking the **Remove selection** button. The component will then disappear from the list box.

More than one component may be removed in one operation. Select all the components required whilst holding down the *shift* key(consecutive selection) or *control* key (nonconsecutive selection), and then click the **Remove selection** button. All the available components may be removed simultaneously by clicking the **Remove all** button.

A component may also be removed from the Virtual IO view itself. As described in Section , all components are shown in a small window with a close button on the right. Clicking this button will also remove the component.

4.4.6.5 Component Properties

Some components have properties that can be changed. Properties might include the number of items within a component or the polarity of input or output pins. The properties of a component can be set or changed by opening the **Add/Remove/Edit Component...** menu item from the Virtual IO view popup menu. Select the desired component from the **Added Components** list on the right of the dialog. Then click **Properties**. This will open the **Edit Component Properties** dialog. The contents of this dialog is different for each component and is described in the section relating to that component later in this chapter.

More than one component may be customized at the same time. Select all the desired components from the **Added Components** list on the right of the dialog whilst holding down the *shift* key (consecutive selection) or *control* key (nonconsecutive selection),. After clicking **Properties**, the **Edit component Properties** dialog will open with one tab for each component. Configure each component on every tab pane, then click **OK**.

4.4.6.6 Wiring Components

After components have been added to the Virtual IO view, they can be connected.

The wiring dialog can be opened by selecting **Edit Wiring...** from the Virtual IO view popup menu, or by clicking on the **Edit Wiring** button after having selected the **Add/Remove/Edit Com-**

ponent. . . menu item from the Virtual IO view popup menu. There must be components added before any wiring can take place.

A connection is made by first selecting a pin of one device and a pin of another device. All added components listed in two scrollable text boxes on the left and right of the dialog. Select the desired component from either list. This list is searchable. To find a known component quickly, start typing the name of the component into the text field above the list box. The first component whose name matches the search string is selected in the list box. Continue typing letters of the component's name until the required component is selected. The search string is not case sensitive.

With a component selected, the pins associated with that component are displayed in scrollable text box under the components. The pins' names are display here, along with symbols to indicate the type of the pin. The symbol showing an arrow pointing toward the pin is an input pin. The symbol is coloured green if the pin is able to be connected. Some pins do not need connection. Such pins include the components power supply or peripheral pins which have not been implemented in HI-TIDE. These pins have greyed out names and pin symbols. The other pin types represented are output pins, and bi-direction pins which can be either input or output. Select the desired pin from the list.

Select the device and pin from the other lists in the dialog. Clicking Connect will make the connection which will then listed in the **Connections** list. A connection is displayed with a form:

```
device_name.pin_name - device_name.pin_name
```

The connection will not be allowed if the selected pins are incompatible, e.g. if both are inputs or both are outputs. Bi-directional pins may be connected to either inputs or outputs, however the code associated the the peripheral must ensure that the pin is set up in the desired state.

More than one pin can be connected with the same operation. Select the pins from each selected component using the *shift* key (consecutive selection) or *control* key (nonconsecutive selection), then click **Connect**. The number of pins selected for each component must be the same otherwise no connections will be made. Connections are made from the first (top to bottom) selected pin in one pin list to the first selected pin in the other list, then the second pin in each list, etc. The pin pairs must be compatible otherwise *no* connections will be made.

If multiple pins are to be connected, but the order in which the pins should be connected is the reverse, click **Connect Reverse Order** after selecting the pins in the usual way. For example, if pins `A` and `B` (top to bottom in that order) have been selected for component `one`, and pins `X` and `Y` (top to bottom in that order) have been selected for component `two`, selecting Connect would make the connections:

```
one.A - two.X
one.B - two.Y
```

but clicking Connect Reverse Order would make the connections:

```
one.A - two.Y
one.B - two.X
```

If more devices need to be added, the button Edit Devices will take you to the **Edit IO Components** dialog showing the **Select Component** display after first saving any connections made.

4.4.6.7 Peripheral Components

The following describe the individual components in detail. More than one of the same component, except for the simulator component, can be added to the Virtual IO view. If more than one component of the same type has been added, they can be configured independently. The options for each component can be specified by selecting the component from the **Added Components** list in the **Edit IO Components** dialog and clicking the **Properties** button.

8051simulator The 8051simulator is considered a peripheral component. Adding a simulator is not mandatory, although is typically done. The simulator component represents the microcontroller being simulated by the debugger.

The available pins on the simulator correspond to the pins of the microcontroller. Not all pins can be wired, e.g. the power supply pins and crystal oscillator pins are assumed to be wired in such as way that the microcontroller would operate normally. Refer to your 8051 datasheet for more information.

The simulator does not have a graphical representation in the Virtual IO view, nor are there any options that can be specified for this view.

LCD Display (Liquid Crystal Display) The LCD Display simulates a LCD with a standard 14 pin IDC connector and LCD controller, such as the Hitachi HD44780. Figure 4.18 shows the **Edit Components Properties** dialog for the LCD panel. The default options and graphical display can be seen.

The options are as follows.

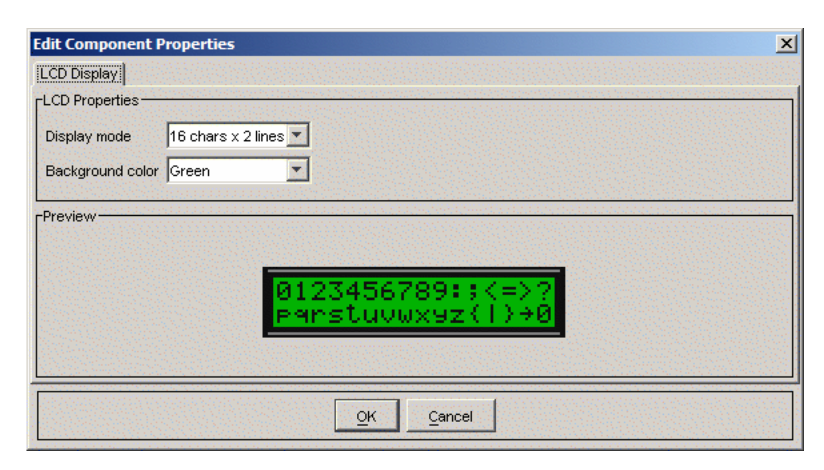
Display Mode Specifies the size and configuration of the LCD panel. The number of characters wide, and the number of rows can be specified.

Background Colour Specified the background colour of the LCD panel.

The LCD panel specifies twelve pins. The power pins are assumed to be pre-wired and do not appear in the pin list. The pins' operation is described as follows.

Contrast This input is not connected.

Figure 4.18: LCD properties dialog



RS (register select) This input pin specifies whether the current read or write cycle is accessing data in the internal RAM of the LCD (high) or is a control operation (low).

R/W (read / write) This input determines if the current memory access is a read (high) or write (low) cycle.

E (enable) This input acts as a data strobe. A high-to-low transition indicates that the data bus is valid.

DB0... DB7 Bidirectional data bus pins.

The 8051 simulator does *not* simulate the data setup and hold times associated with the LCD memory interface.

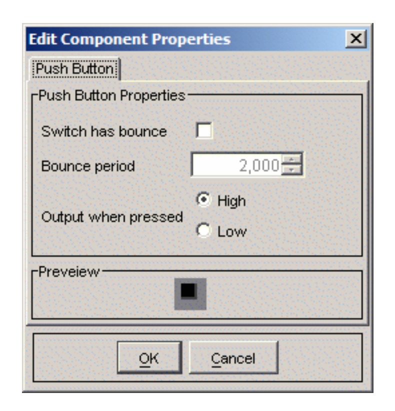
LED Panel The LED panel simulates a bank of one or more light-emitting diodes. Figure 4.19 shows the **Edit Components Properties** dialog for a bank of four LEDs in a panel. The following options are available.

Number of LEDs This specifies the number of LEDs that will appear in the panel. Up to 32 LEDs may be specified.

LED colour This allows selection of the LED colour.

Figure 4.19: LED properties dialog

Figure 4.20: Push button properties dialog



Vertical orientation The LED panel can be shown with the LEDs aligned vertically by checking this option.

Active low inputs With this checkbox in the default unchecked setting, the diode has its cathode connect to ground and the anode as input. If a high voltage is applied to the diode's input, the LED illuminates. Enabling this checkbox connects the diode's anode to the positive supply rail and the cathode becomes the input. If a low voltage is applied to the diode's input, the LED illuminates.

LED panels have one input pin for each diode in the component. The input is either the anode or cathode of the diode, and the other pin associated with each diode is connected to either the ground or positive supply, respectively, depending on the diode's property settings.

Push Button The push button simulates a single momentary switch. Figure 4.20 shows the **Edit Components Properties** dialog for the push button. The following options are available.

Switch has bounce Enabling this checkbox causes the output voltage of the switch to "bounce" with each press, as would be expected with a mechanic switch. The bounce period can be specified in the **Bounce period** spin box. This period is specified in instruction cycles.

Output when pressed This option specifies the output voltage when the switch is pressed. Changing this setting changes the virtual wiring for the switch. Selecting High will result in the output pin of the switch normally at a low voltage, but will become high whilst pressed.

Push buttons have one output pin.

Chapter 5

HI-TIDE Projects

HI-TIDE encapsulates various aspects of an application being developed. This information is called a *project* and is saved on disk as a *project file*. The state and views of the project are part of the information saved to disk and are restored upon reloading that project.

This chapter explains what information is stored in a project and how to create, open and manage projects within HI-TIDE.

5.1 Toolsuites

A *toolsuite* is a set of HI-TECH components with which projects can be created, built and executed. A toolsuite typically includes the following tools.

- Compiler options and driver
- Debugger options and drivers
- Code wizard options

A project file is specific to both a toolsuite and the toolsuite's version.

The components of a toolsuite are contained in files that are shipped with a compiler package and a toolsuite has the same version number as the compiler with which it is distributed.

As more recent versions of a toolsuite are installed, a HI-TIDE project can be updated to make use of the new toolsuite, see Section 5.6.1 for information on changing toolsuite versions. A new toolsuite version may contain new devices that can then be selected, new compiler options, new features in the Code wizard or even totally new debuggers and debugger options. Once the project file has been updated to use the new toolsuite, these new features will become available via the

menus and dialogs. A project converter is automatically run when you update toolsuite version to maintain the project file, but as new options may be available, the options associated with all the components should be reviewed.

It is also possible to change the toolsuite associated with a project, e.g. a project set up for HI-TECH C for MSP430 could be changed to HI-TECH C for ARM. No project conversion takes place when changing toolsuite, but those settings which are common to both toolsuites will be preserved across the change. See Section 5.6.1 for more information. With a different toolsuite set up with a project, you can then select a different family of devices and debuggers available for these devices.

5.2 Project Information

HI-TIDE's project files can be given any file name with the extension `.hprj`.

A project keeps track of all the files that are associated with the application, as well as what options are selected. Information specifying the graphical layout of the views are also saved in the project. Also saved in the project file are the tools used with the project. This includes the toolsuite, its version, compiler and debugger used.

Some configuration information is not considered to be part of a project. This information relates to general preference options for HI-TIDE. This information is stored in separate files in the .hitide directory placed in the home directory of the user. It specifies such things as the general preferences, the size of the HI-TIDE window, its relative screen position, recently opened files and plugin settings.

5.3 Creating A New Project

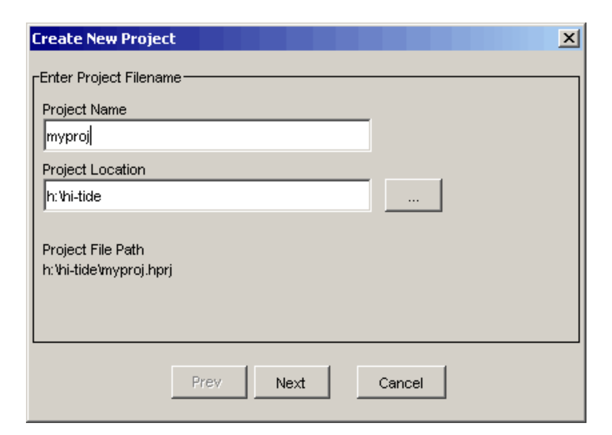
To create a new project, select **New Project** from the **Project** menu. The *Project wizard* will then be displayed. The Project wizard will present a series of dialogs collecting the information needed to create a new project. If there was a project opened prior to selecting the menu item, it will be saved (depending on the preferences set - see Section 2.3).

When the project is created, it will then be displayed in HI-TIDE. The new project will not be saved to file until it is explicitly saved by selecting Save Project from the Project menu.

5.3.1 Project wizard

The Project wizard is broken up into seven screens *project filename*, *project toolsuite*, *target device*, *target device package*, *compiler*, *debugger* and *project source files*.

Figure 5.1: Project wizard — project details



5.3.1.1 Project Filename

The project filename screen is where the name of the project file and the directory is set. The *Project* Name textfield is where the name of the project is specified. The **Project Location** text field is where the directory of the project is specified. The complete path of the project filename is shown underneath in the **Project File Path** field. The path of the project file will be updated as the name or directory is entered. See 5.1.

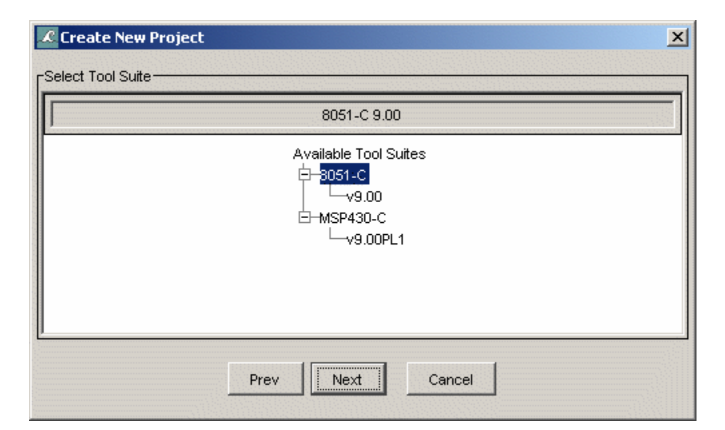
When the project file is saved, if the name of the project file does not have the `.hprj` extension, HI-TIDE will automatically append this extension to the filename.

The **Project Location** field specifies the directory where the project file and other compiler generated files are to be placed. The directory can be entered manually or can be selected via the directory chooser by clicking on the **Browse Directories** (“**...**”) button.

The name of the project must be filename only and not a path. The file name must not include any of the following characters: `! @ # $ % ^ & * ( ) - + = | , < > : ; { } [ ] / \`

When a filename is entered, the **Next** button will be enabled. When the next button is selected, the Project wizard will check the validity of the filename. If the filename is not a valid filename, the Project wizard will notify the user. If the filename points to a file that already exists, the user will be asked whether the file should be overwritten.

Figure 5.2: Project wizard – toolsuite selection



5.3.1.2 Project Toolsuite

The project toolsuite selection screen is where a toolsuite to be used for the project is selected.

The left column of the project toolsuite screen, labelled *Supported Tools*, contains the toolsuites available for selection. Selecting a toolsuite from the list will populate the versions list, labelled *Supported Versions*, on the right hand side. The versions list shows the versions of a toolsuite that are installed and are available for selection.

The Supported Versions list only displays the toolsuite version supported by that version of HI-TIDE. A user may have more toolsuites installed that what is displayed, but those versions not displayed are not supported by HI-TIDE and should not be used.

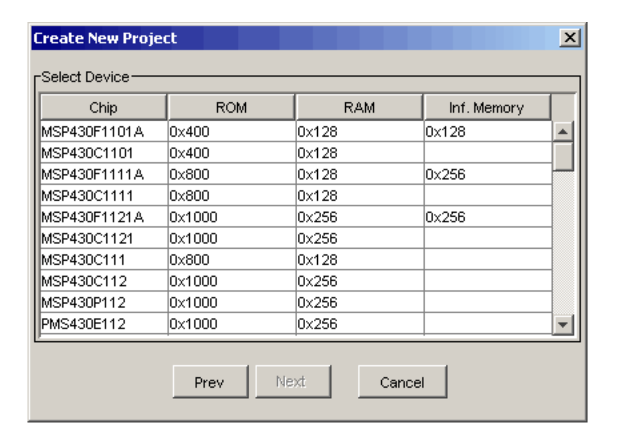
A typical project toolsuite selection screen is show in Figure5.2.

Although a toolsuite is required to be selected to create a new project, it can be changed at anytime once the project is created. See section 5.6.1 for more details on changing the project toolsuite.

When a toolsuite is selected from the *Supported Tools* list, the Project wizard will automatically highlight the latest version of the toolsuite in the *Supported Versions* list.

Selecting an item from the versions list will enable the **Next** button. Clicking on the **Next** button will prompt the Project wizard to proceed to the next screen. Clicking on the **Prev** button will return to the previous screen in the Project wizard.

Figure 5.3: Project wizard — target device



5.3.1.3 Device Selection

The device is the chip that is to be used for the project. Although the device must be selected to create a new project, it can be changed at any time once the project has been created. See Section 5.6.2 for more details on changing the project target device.

The device selection screen contains a table with the chips. The chips shown in the list are the chips available and supported in the version of the selected toolsuite. The data shown in the device selection screen may differ depending on specific toolsuite, but will mainly consist of the name of the chip, its manufacturer (if the chips are produced by more than one manufacturer) and its various device memory. Figure 5.3 shows a typical target device selection screen.

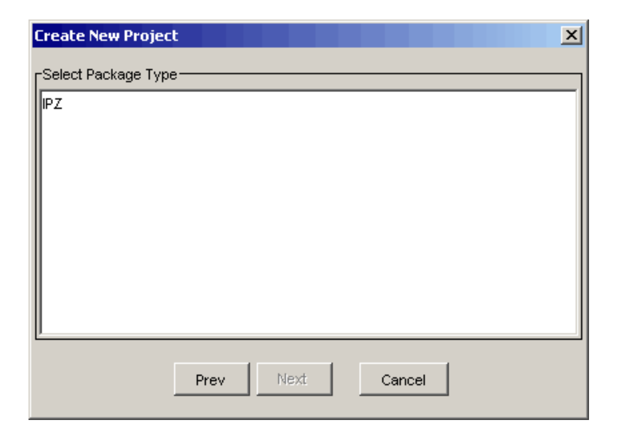
When a chip is selected, the **Next** button will be enabled. Clicking on the **Next** button will proceed to the next screen in the Project wizard. Selecting the **Prev** button will return to the previous screen in the Project wizard.

5.3.1.4 Device Package

The device package selection screen is used to select the packaging type of the target chip. The chip package may affect things such as the *code wizard* or the simulator as the chips outputs may map to different pins in different packages. Although a chip package must be selected to create a new project, it can be changed at any time. See Section 5.6.3 for more details on changing the device package types.

The device package selection screen consists of a list of the packages available for the selected

Figure 5.4: Project wizard — device package



target device. Figure 5.4 shows a typical target device selection screen.

When a package type is selected, the **Next** button will be enabled. Clicking on the **Next** button will proceed to the next screen in the Project wizard. Selecting the **Prev** button will return to the previous screen in the Project wizard.

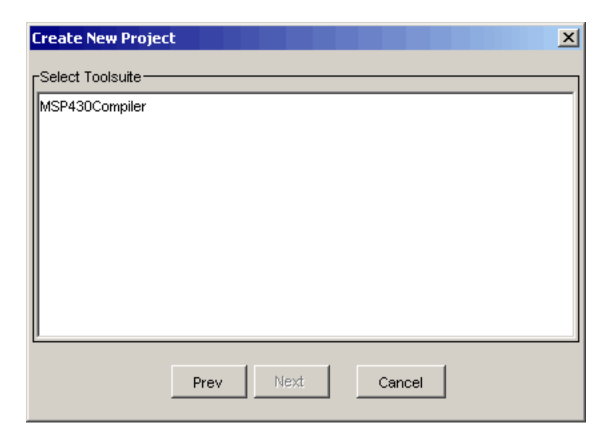
If there is only one device package type, it will by automatically highlighted and the Next button will be enabled as a result..

5.3.1.5 Compiler Selection

The compiler selection screen allows the user to select the compiler from the toolsuite to be used with the project. The compiler is closely tied in with the toolsuite and cannot be changed without changing toolsuite. For example if a project was using a compiler version 9.0 and was to be compiled using version 9.20, the toolsuite will have to be changed to version 9.20 to use the compiler from that version. Figure 5.5 shows a typical compiler selection screen.

If there is only one compiler available for selection from the toolsuite, the Project wizard will automatically select the compiler and skip the compiler selection screen, This will

Figure 5.5: Project wizard — compiler selection



be the case with most toolsuites.

5.3.1.6 Debugger Selection

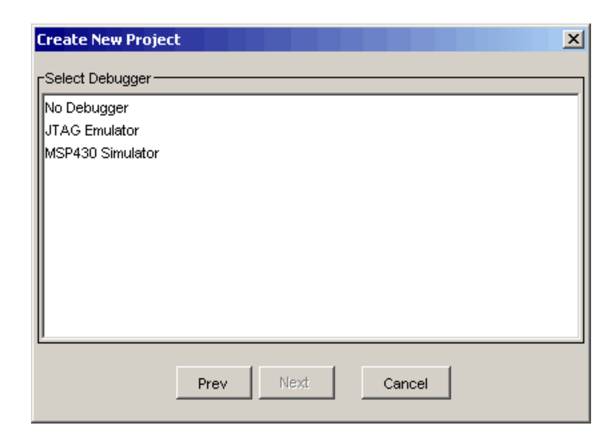
The debugger selection screen allows the selection of the debugger to be used with the project. A debugger must be selected for the project, but can be changed at any time, once the project is created. Refer to Section 5.6.4 for more information on changing the project debugger.

The debugger selection screen is similar to both the package type selection and compiler selection. Figure 5.6 shows a typical debugger selection screen.

The debugger list shows the debuggers that are available in the selected toolsuite for the selected target device. Some toolsuites (especially later versions) may have more debuggers available than others. The debuggers shown are also affected by the selected device. A debugger may support one device over another.

At the least, the list of debuggers will always contain the item No Debugger. Selection of the No Debugger will set the project to, not having a debugger and the debugger related menu items and toolbars will be disabled.

Figure 5.6: Project wizard — debugger selection



When a debugger (or *No Debugger*) is selected, the **Next** button will be enabled. Clicking on the **Next** button will proceed to the next screen in the Project wizard. Selecting the **Prev** button will return to the previous screen in the Project wizard.

5.3.1.7 Project Source Files

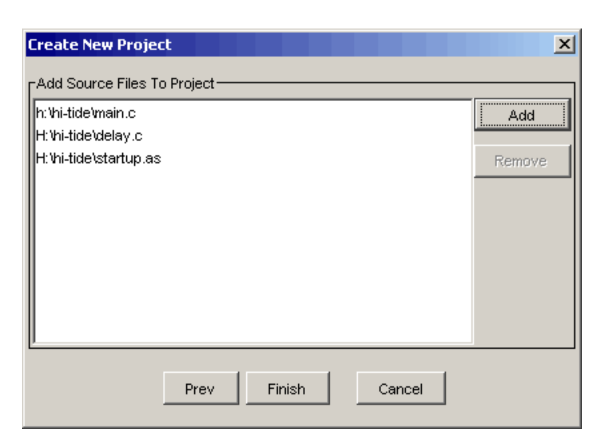
The project source files selection screen is to allow the quick addition of source files into the project. This step is not necessary in the creation of the project, and hence the **Finish** button is enabled without having to add any files.

To add files to the project, click on the **Add** button. This will display the file chooser dialog for selection of files. Files can be selected one at a time or multiple files can be selected from this dialog. When the files are selected from the file chooser dialog, they will be added to the list of source files in the project source files screen. The order in which the files are added will be the order that they appear in the list. Figure 5.7 shows a typical source files selection screen with some files added.

To remove files from the list, select the file to be removed. This will enable the **Remove** button. Clicking on the **Remove** button will delete the file from the source files list.

The order in which the files appear in the source files selection screen is the order in which the files will be added to the project. This will also be the order that they appear in the project view. This is also the order in which the files are compiled and linked.

Figure 5.7: Project wizard — source file selection



As it is not necessary to select source files in the creation of a new project, the **Finish** button is always enabled. Clicking on the **Finish** button will close the Project wizard and the new project will be created and opened in HI-TIDE. Clicking on the **Prev** button will return to the previous screen in the Project wizard.

5.4 Managing Projects

5.4.1 Opening Existing Projects

An existing project can be opened by selecting **Open Project...** from the **Project** menu. When this menu is selected a file dialog will be shown allowing a project file to be selected.

A recently opened project can be opened quickly by selecting from the **Recent Projects** submenu in the **Project** menu. The submenu will display the paths to a number of project files that have been recently opened. Selecting a project file will open that project in HI-TIDE. The number of files stored in this menu can be configured in the **General Preferences** dialog.

HI-TIDE, by default, will load the last opened project when starting. This behaviour can be configured in the **General Preferences** dialog as well. See Section 2.3 for more information on setting and changing the preferences.

5.4.2 Saving Projects

To save the state of the project to disk, select **Save Project** from the **Project** menu. Selecting the **Save** All menu item from the **File** menu or **Save All** button from the standard toolbar will also save the project file, as well as all opened editor files.

To save the opened project file under a different filename and/or directory, select the **Save Project As** from the **Project** menu. When the **Save Project As** menu is selected a file dialog will be shown which will allow a new file name and/or directory to be selected. This will change the name of the project currently opened to the new name. The title of the project in the HI-TIDE window will also be updated to show this change. The output node in the project view will also be updated.

5.4.3 Closing Projects

To close a project, select **Close Project** form the **Project** menu. Opened projects will automatically be closed when exiting HI-TIDE.

Saving of project files when a project is closed can be configured in the **General Preferences** dialog. By default if the project has been modified and is about to be closed, a prompt will appear requesting if the project file should be saved. Other settings are a project file should always be saved when it is closed or a project file should never be saved when it is closed.

5.5 Managing Project Source Files

The managing of project source files, and library files, can easily be done via the project view or file menus. Through the project view, the files are mainly managed by right-clicking on the file or folder and using the associated popup menu.

The options available include adding and removing source files from the project, compiling of files to intermediate files and setting of the compiler options for the files. These functions are described in detail in the following sections.

5.5.1 Adding Files To The Project

There are several methods of adding files to the project. The files that can be added to a project include C or assembler source files (.c or .as), library files (.lib) and object files (.obj). Source files can be existing or new files can be created and added. The methods of adding source files are described in the following.

A project cannot have two or more files (source or object) with the same filename (not including extension). When adding a file, if a file with the same name is already in the

project exists, the second file will not be added and HI-TIDE will issue an error

---

The first method is the addition of existing source files to the project via the source file selection screen in the Project wizard, when creating a new project (see Section 5.3.1.7). This method adds the source files into the project at the time when the project is created.

Another method of adding existing source files to the project is through the use of the popup menu in the project view. Right clicking on the *C Files* folder or *Assembler Files* folder will show the popup menus for those folders respectively. Existing C or assembler files can be added by selecting **Add** Existing C File(s)... or **Add Existing Assembler File(s)...** menu items from the respective popup menus. This will open a file dialog to select any number of files. The files will then be added in to their corresponding folders. When using the **Add Existing C File(s)...** option, non-C files cannot be added. Similarly, when using the **Add Existing Assembler File(s)...** option, non-assembler files cannot be added.

Addition of existing source files can also be done through the **Project** menu, using the **Add Files To** Project... menu option. This is similar to right-clicking and using the file folder popup menus, except that it is not file specific like the folders. Both C and assembler files can be selected from the file dialog and added. HI-TIDE will sort the file into their appropriate folders.

Alternatively, if a file is opened in the file editor, and a file of the same name is not already included in the project, the file can be added to the project by selecting the **Add File To Project** option from the **Project** menu. This menu item will only be enabled if the file is valid for addition to the project.

A new source file can be created and added to the project in one action by the file folders' popup menu. If a new C file is to be created and added to the project, right-click on the **C Files** folder and select the popup menu item **Create And Add New C File**. This will open a file dialog, prompting for the filename to save the new file as. Selecting **Save** in the file chooser dialog will create the file and add that file to the project. The file will also be opened in an editor view and added to the workspace in a new workspace tab labelled as the filename.

Existing object or library files can be added to their respective file folders by right-clicking on the **Object Files** folder or **Libraries** folder. Selecting the **Add Existing Object File(s)...** menu item or **Add Existing** Library File(s)... menu item (from the respective popup menus) will open a file dialog for selection of the existing object files or library files. The file dialog supports selection of one or more file at a time, but the files must be of the correct type - i.e. only .obj files can be selected for object files and .lib files can be selected for libraries. Clicking the **Open** button in the file dialog will import the selected file(s) to the project.

With all of the file folders, pressing the Insert key on the selected file folder will open the file dialog to add existing files of the folder type. This is the same as the selecting the **Add Existing ...** file type popup menu option for each of the respective folders.

5.5.2 Removing Files From The Project

To remove one or more files from the project, select the file that are to be removed from the project view by right-clicking on it and select **Remove From Project** from the popup menu.

To select more than one file at a time hold down the *control* key (for non-contiguous selection) or Shift key (for contiguous selection) when selecting the files. Right-clicking over the files after they have been selected will trigger the popup menu. Select **Remove From Project** to remove the selected files.

•

The file folders cannot be removed from the project. Right-clicking on the folders only allow files to be added.

Files can also be removed by pressing the Delete key on the selected file.

5.5.3 Changing Compiler Options

Compiler options can be set on a global basis or per file basis. Global options are handled by the output file and apply to all files that do not have *file specific options* set. Global compiler options are changed through the ***Global Compiler Options*** dialog. This dialog can be opened by double clicking on the output node in the project view or by right-clicking on the output node and selecting **Global Compiler** Options... from the popup menu. The ***Global Compiler Options*** dialog can also be displayed by selecting **Global Compiler Options...** from the **Project** menu.

Project source files can have compiler options that are specific to the file and different to the global compiler options. These options are referred to as *file specific options*. By default when a C or assembler file is added to the project, global compiler options are applied to the file and used for that file when it is compiled. To change the options of a file to be file-specific, right-click on the file node in the project view and select **File Specific Options** from the popup menu. This will show the ***File Specific Options*** dialog, which is very similar to the ***Global Options Dialog***, where the options can be set for that file only. To switch between using file-specific options and global options, select or deselect the "*Use custom* options" checkbox in the ***File Specific Options*** dialog.

5.5.4 File Properties

HI-TIDE can display the properties of files in a project in the *File Properties* dialog. The properties tah are displayed include the filename, file path, file size and when a file was last modified.

To open a ***File Properties*** dialog, right-click on the file in the project view and select **Properties** from the popup menu of that file node. Figure 4.2 shows a typical ***File Properties*** dialog for a file.

5.5.5 Dependency Files (Header Files)

Dependency files are automatically handled by HI-TIDE. Dependency files are files, usually header files, that are #included into another file. When a file is added to the current project, or it is compiled, HI-TIDE does a scan for the dependencies of this file, and the dependencies are added to the project.

To view the dependencies of a particular file, right-click on that file node in the project view. If the file has dependency files, the **Open Dependency** menu item in the popup menu will be enabled. If the file does not have dependency files, the menu item will be disabled. The **Open Dependency** option is a sub-menu, which expands to list the dependency files of that particular file. Clicking on a dependency file will open the file in an editor view in a new workspace tab. The workspace tab will be labelled with the name of the file.

5.6 Changing Project Settings

When a project is created, certain options were selected in order to create a project. These included the toolsuite, the target device to be used, the package type of the target device and the debugger. Once the project is created, all these options can be changed. The following describe how to change these settings.

5.6.1 Changing Toolsuite

Toolsuite is an essential part of a HI-TIDE project, and a project must always have a toolsuite set. The toolsuite determines the chips that are available for the project as well as the tools to use to compile and debug the project. A toolsuite must be selected when creating a project, but can be changed at any time.

Once a toolsuite is selected for a project, a toolsuite from a different chip architecture type cannot be selected to replace the original. Only toolsuites of the same architecture type can replace the original. This allows you to change between different versions of toolsuites from the one toolsuite family.

To change toolsuites, select the **Change Toolsuite...** menu item from the **Project** menu. A ***Select*** Toolsuite dialog will appear to allow selection of a different toolsuite. The dialog will be similar in appearance to the project toolsuite selection screen in the Project wizard (see Figure 5.2).

The list of *Supported Tools* list, however, will only display the name of the currently selected toolsuite and it will be highlighted as well. The *Supported Versions* list will display the different toolsuite versions available for selection. The currently selected toolsuite version should be highlighted to indicate that it is currently selected.

Clicking on the **Finish** button will set the newly selected toolsuite as the toolsuite to use (if it has changed). Selecting **Cancel** will cancel the operation and restore the current toolsuite.

•

When a new toolsuite is selected, it may not have support for the currently selected target device, especially if the newly selected toolsuite is older in version number to the currently selected toolsuite. A different target device and device package type may need to be selected as a result. Likewise, the newly selected toolsuite may not support the debugger previously selected. A different debugger may need to be selected. HI-TIDE will notify the user in all cases where the previous selections cannot be restored when a new toolsuite is selected.

5.6.2 Changing Device

A project must always have a target device selected, however, the target device can be changed. This can be done by selecting **Change Device...** from the **Project** menu. This will display the ***Select Device*** dialog, to select a different target, which is similar in appearance to the target device selection screen in the Project wizard (see Figure 5.3). The dialog should display the selected chip at the top of the view. The list of chips available for selection will be the devices supported by the selected toolsuite.

The target device can also be changed by double clicking on the target section of the status bar (see Section 3.3). This will also display the ***Select Device*** dialog.

•

The package type for the current target device may not be supported in the new device. Selecting a new device may also require selection of a package type for the new device. HI-TIDE will notify the user of this requirement. Likewise, the debugger selected for the current device may not support the new device. A different debugger selection may also be required on selection of the new device. HI-TIDE will also notify the user of this requirement.

Clicking on **Finish** will set the newly selected chip as the target device (if it has changed). Clicking on **Cancel** will restore the current device.

5.6.3 Changing Device Package

Some tools can be affected by the device package type. An example could be the simulator, where it simulates the output of the device, and the output may map to different pins, depending on the

package type of the target device. Target package types can be changed to accommodate such needs when dealing with chips that have different package types.

To select a different package type, select the **Change Package...** menu item from the **Project** menu. The ***Select Device Package*** dialog will appear to allow the selection of a different package. The dialog will be similar in appearance to the target device package selection screen in the Project wizard (see Figure 5.4).

Clicking on **Finish** will set the newly selected device package type (if it has changed). Selecting **Cancel** will restore the current package type.

5.6.4 Changing Debugger

To change debuggers, select the **Change Debugger...** menu item from the **Project** menu. This will display the ***Select Debugger*** dialog. The dialog is similar in appearance to the debugger selection screen in the Project wizard (see Figure 5.6).

Unlike the toolsuite, target device or device package type, a debugger does not need to be set for the project. This can be set by selecting "*No Debugger*" from the debugger list.

The debugger can also be changed by double-clicking on the debugger name of the status bar (see Section 3.3). This will display the ***Select Debugger*** dialog.

Clicking on **Finish** will set the newly selected debugger as the debugger to use. Clicking on **Cancel** will restore the current debugger.

Chapter 6

C-Wiz — The Code Wizard

An advanced feature integrated with the HI-TIDE toolkit is C-Wiz, the Code wizard. This is a graphical development tool designed to minimize the burden associated with setting up microcontrollers and their on-board peripherals. Instead of searching through manufacturer's data sheets to learn the bit manipulations needed by each peripheral in order to get it to work, you can simply run the Code wizard and let it do the job for you. The dialog lets you select which peripherals you intend to use and describe how you would like each one to operate. The Code wizard translates these settings into corresponding C code that can be executed on your selected target device to set up the peripherals to the given specifications.

6.1 Starting the Code Wizard

As the Code wizard is a plug-in tool to HI-TIDE, it can be started by clicking the **Code Wizard** button in the user-tools toolbar. Alternately, the **Tools** menu contains an item called **Code Wizard**, selecting this item will also open the **8051 Code Wizard** dialog. Figure 6.1 shows the HI-TIDE dialog indicating both methods to start the Code wizard.

If the button and menu item are deactivated, this indicates that the Code wizard doesn't support the microcontroller selected in the HI-TIDE project.

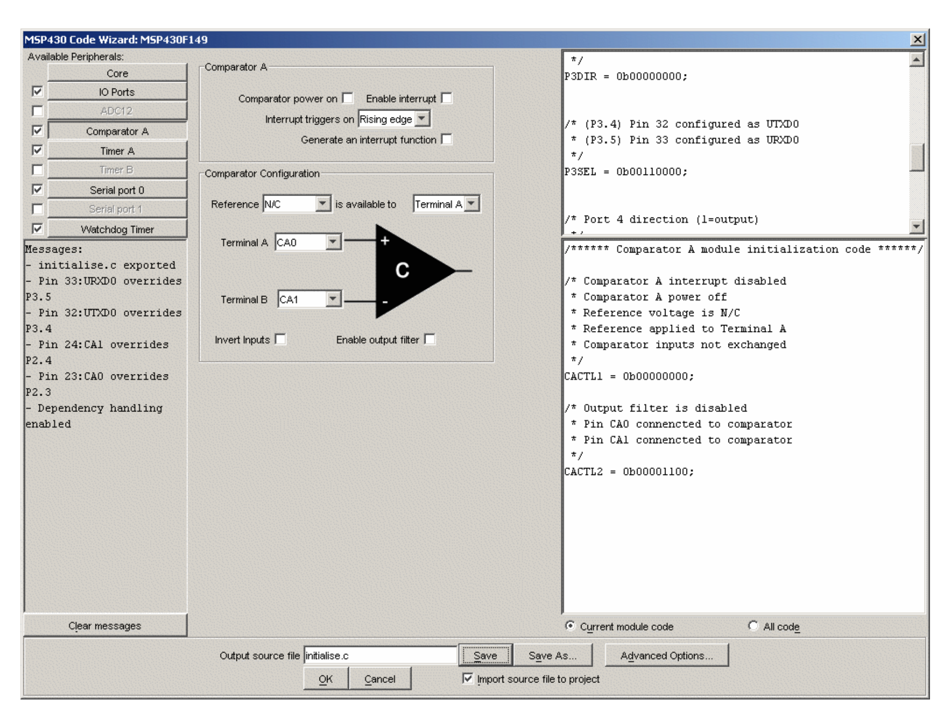
6.2 The 8051 Code Wizard Dialog

The graphical interface of the Code wizard is composed of five smaller panels inside a dialog. A typical screenshot of dialog is given in Figure 6.2.

The panels which make up the **8051 Code Wizard** dialog are described as follows:

Figure 6.1: Starting the Code wizard from within HI-TIDE

Figure 6.2: A typical Code wizard dialog



6.2.1 Peripheral Selection Panel

The upper-left-most panel of the **8051 Code Wizard** dialog is the Peripheral Selection Panel. This panel lists all of the configurable peripherals that are available to the microcontroller selected in the current HI-TIDE project.

Each peripheral in this list will have a checkbox to *activate* (indicate the peripheral is to be initialized) and a button used to *select* the peripheral. When a peripheral is selected its configurable settings are displayed in the Configuration Panel.

6.2.2 Configuration Panel

The Configuration Panel occupies the main central area of the **8051 Code Wizard** dialog.

This panel displays the configurable settings available for the currently selected peripheral. Initialization code is generated for each peripheral based on the settings made in the Configuration Panel.

6.2.3 Messaging Panel

The panel to the lower-left of the dialog is the Messaging Panel. The Code wizard will use this panel to report any warnings or messages that result during the generation of initialization code.

In most cases these messages will result from conflicts which arise when multiple peripherals try to access the same resource. Such messages are interpreted in Section 6.9 of this manual.

The newest messages appear at the top of the Messaging Panel. If the list of messages grows too long, it can be cleared at any time by pushing the **Clear messages** button.

6.2.4 Generated Code Display

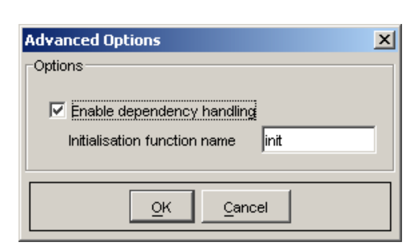
The right-most panel contains a large text area used to display the generated initialization code. The code in this window is dynamically generated and can be seen changing in accordance with adjustments in the Peripheral Selection Panel and Configuration Panel.

Controls are available to switch between viewing the generated code for the whole system or viewing the specific code generated for the currently selected peripheral. A comparison of the different views for the Generated Code Display is shown in Figure 6.6.

6.2.5 Control Panel

At the bottom of the **8051 Code Wizard** dialog is a small panel with controls which allow the generated code to be output to a file. The facility is also available to automatically import this file into the current HI-TIDE project. Also within this panel is the **Advanced options...** button as well as the **OK** and **Cancel** buttons used to exit the Code wizard.

Figure 6.3: The Advanced Options dialog



6.2.6 Advanced Options Dialog

Apart from those options available in the Control Panel the Code wizard has various options that can be configured using the **Advanced Options** dialog. This is shown by clicking the **Advanced Options...** button within the Control Panel of the **8051 Code Wizard** dialog. The layout of the **Advanced Options** dialog is shown in Figure 6.3.

The advanced options that can be configured include:

6.2.6.1 `Enable dependency handling`

This checkbox enables you to toggle the Code wizard's automatic handling of shared resources (dependency handling). This is discussed further in Section 6.9.

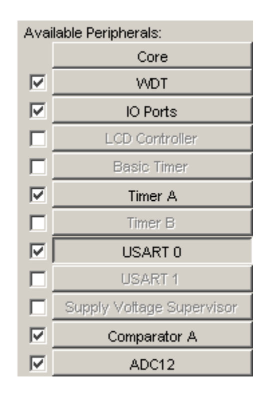
6.2.6.2 `Initialisation function name`

This text field allows you to specify the name of the function the Code wizard will create when saving a file. By default the function name is set to `init`. See Section 6.6 for more information about saving to files.

6.3 Selecting Peripherals

Microcontrollers can contain a large number of internal peripherals, so it would be unnecessary and wasteful to generate initialization code for peripherals that you don't intend to use. For this reason the Code wizard allows you to select which peripherals you require code to be generated for. This is done via the Peripheral Selection Panel by ticking the checkbox associated with each peripheral. As this box is ticked, its corresponding selection button will be activated. Pushing the selection button

Figure 6.4: Peripheral selection panel of C-Wiz



will select the peripheral so that its setting are displayed in the configuration window. The Peripheral Selection Panel is illustrated by Figure 6.4.

In this figure, `USART 1` is a peripheral that has been deactivated. Its checkbox is clear and its *selection* button deactivated. No code will be generated for `USART 1`. `Timer A` has been activated so initialization code will be generated for this peripheral. `USART 0` is activated and selected, this means that code will be generated for `USART 0` and its settings will presently be displaying in the **Configuration** Panel.

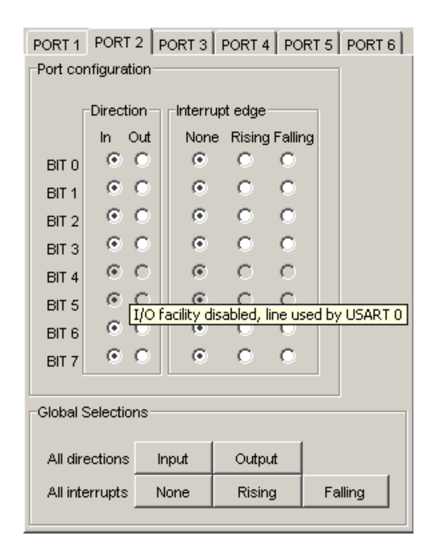
It is important to understand that if a peripheral is deactivated in the Code wizard, that does not guarantee that the peripheral will be inactive in your target system. If a peripheral receives no initialization code it will operate in its default state, which in some cases might be active.

Code generation for each peripheral can be activated or deactivated at will, with the exception of the Core module. Every microcontroller supported by the Code wizard will have a configurable module called **Core**. Initializing the Core module is required for setting up system-critical attributes such as oscillator settings, global interrupt control or memory configuration.

6.4 Configuring Peripherals

As mentioned earlier, the Configuration Panel of the Code wizard can present the initialization options for each of the device's peripherals in turn. To configure initialization code for a selected peripheral it must be activated (via the checkbox in the Peripheral Selection Panel) and selected. The Configuration Panel only presents the settings that are available for the peripheral that is currently selected.

Figure 6.5: Typical I/O port configuration panel



When the Configuration Panel display is displaying settings for a particular peripheral, it is only a matter of adjusting the selection components to describe your system's needs. A brief description for each setting will appear in a floating text box if the mouse is left to float over a selection component for more than a second. If a selection component is deactivated or unavailable, the floating text box will give the reason why the setting is unavailable. In Figure 6.5, the floating text box explains that all selections for I/O Port 2, bit 5 are unavailable because this I/O line is already in use by USART 0 (see Section 6.9 for information on shared resources).

As each setting is adjusted, the consequential code is automatically updated in the Generated Code Display.

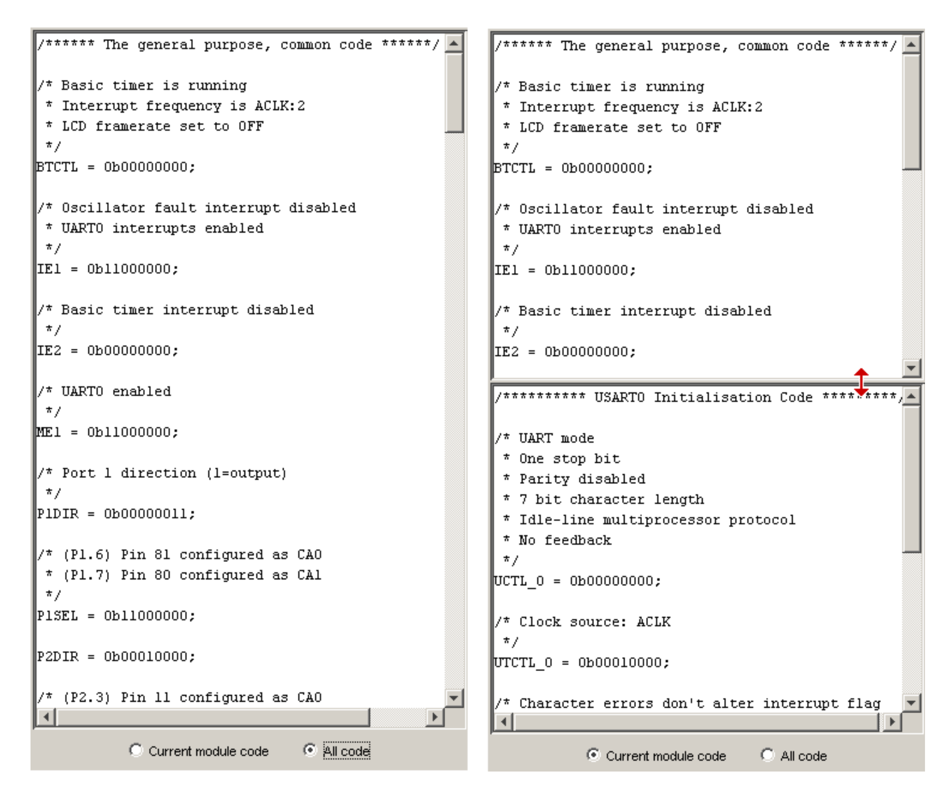
This process can be repeated for all of the peripherals that require to be initialized.

6.5 Viewing Generated Code

Program code generated by the Code wizard is shown in the Generated Code Display. Changes made in the Configuration Panel will be dynamically updated in the Generated Code Display.

Generated program code is fully commented so that the selected modes/settings encoded with each instruction can be clearly identified. This also makes for easier program maintenance if there

Figure 6.6: Comparison of generated code display modes

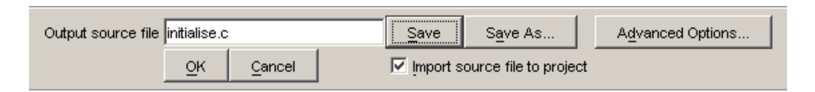


is a need to modify the initialization code at a later date.

Radio button controls below the text area are used to select how the generated code will be presented. If view is set to **All code**, one large *code view* area will show the commented initialization code for all of the activated peripherals. A scroll bar can be used to roll the view over the entire block of generated code.

If the **Current module code** option is selected, the output display will split into two code views. The code view shows initialization code that affects multiple or all peripherals (*common code*). The lower code view shows initialization code that is specific to the currently selected peripheral. The border between the peripheral and common code views can be dragged by the mouse to adjust the ratio between the two views. These two views are compared in Figure 6.6.

Figure 6.7: Control panel of C-Wiz



6.6 Saving to Files

If it is desired to save a copy of the generated initialization code to a file, this can be done via the Control Panel. To save the generated code to a file specify the path and file name in the **Output source file** text field, then press the **Save** button. To search your file system for a specific directory, press the **Save As...** button. This opens a dialog box to allow you to search your file system to find a suitable directory to save the generated code to.

If the **Import source file to project** checkbox is ticked when either the **Save** or **Save As...** button is pushed, the source file named in the text box will also be imported to the current HI-TIDE project.

The saved file will contain the initialization code and comments for all peripherals that have been activated in the Code wizard. This is true even if the Generated Code Display has been set to view only the code for the current peripheral. When the code is saved, the code wizard will create a function definition, with a name specified in the **Advanced Options** dialog (see Subsection 6.2.6), which will contain the generated code. A small sample of the generated code is as follows:

```
#include <msp430.h>
/* Initialisation code generated for the HI-TECH Software
   MSP430 C compiler by the HI-TIDE Peripheral Wizard */
void init(void)
{
/****** The general purpose, common code ******/
/* Oscillator fault interrupt disabled
 * UART0 interrupts enabled
 */
IE1 = 0b11000000;
/* Basic timer interrupt enabled
 * UART1 interrupts enabled
 */
IE2 = 0b10110000;
/****** System Clock initialization code ******/
/* Global interrupts are enabled */
```

```
_BIS_SR(GIE);
/****** Watchdog module initialization code ******/
/* WDT operates in Off mode
 * WDT interval is tSMCLK x 64
 * NMI pin causes reset
 */
WDTCTL = 0b0101101010000000;
/********** Basic Timer initialization code **********/
/* Initial value on timer 1 is 0
 * Initial value on timer 2 is 0
 */
BTCNT1 = 0b00000000;
BTCNT2 = 0b00000000;
}
```

Inside the function, the initialization code required for each peripheral is quite separate. Code for each different peripheral is introduced with a large identifying comment. This makes it easier to separate specific code if it is intended to isolate individual peripheral segments to independent functions or modules.

The only exception to the rule is in the case where a single instruction may have an effect on multiple peripherals. An example of this is as follows:

```
/* Basic timer interrupt enabled
 * UART1 interrupts enabled
 */
IE2 = 0b10110000;
```

This is a typical case where a single register is responsible for controlling attributes (in this case interrupts) for several peripherals. Since such code can't be solely associated with any single peripheral, it is classified as *common code* and will appear in a separate section to be found at the beginning of the function.

It is important to remember at all times that the file saved to is a generated file. This is relevant because any modifications you make to the file later will be lost if you later ask the Code wizard to save the file again. In this event, the Code wizard will simply re-generate the code from the peripheral settings it has been given and overwrite the specified file. It has no knowledge of any changes that you have made to the file since last time. Take care and be aware of this consideration if you are modifying the generated file.

6.7 Accessing the Initialization Code

If the Code wizard is instructed to save its generated code, it will be saved to the specified file in a function with the name specified in the **Advanced Options** dialog (see Subsection 6.2.6). It is likely that this file will ultimately be included as an additional file in your HI-TIDE project. If so, any module that would call on this function will require a function prototype such as this:

```
extern void init(void);
```

It is recommended for your program to call this function very early in its execution, so that the device and its peripherals are ready before any peripherals are accessed and before any system events such as interrupts or watchdog timeout. The function call would look like:

```
init(); // Initialize device and peripherals
```

Upon return from this function call, the microcontroller and its peripherals will have been set to the modes specified in the **8051 Code Wizard** dialog. It is not a strict requirement for this function to be called `init` so if you choose to rename the function, also adjust the function prototype and function calls accordingly.

6.8 Generating Interrupt Service Routines

Some peripherals have the ability to trigger an interrupt on a particular condition or specific event. If so, the Code wizard can generate a function template to enable your program to service this interrupt. To generate an interrupt service routine, simply tick the checkbox in the peripheral's settings corresponding to generating an interrupt function. Some peripherals can have multiple interrupt routines, each triggered on a different event. In this case there may be a separate checkbox to generate each different interrupt function.

It is important to realize that there is a difference between selecting to *enable* an interrupt and selecting to *generate* an interrupt function. Enabling an interrupt sets up the device to trigger an interrupt (and go to an interrupt function) upon the trigger condition being satisfied. It is quite valid to have an interrupt function without enabling the interrupt - you may want to enable it later in your program, but it is unwise to enable an interrupt trigger without also providing an interrupt function to service it.

Of course the Code wizard has no idea of what you want your interrupt service routine to do, so it will only generate a template of the interrupt function. In this way it sets up vectoring to the function automatically, so all that is left for you to do is fill in the content of the interrupt service routine in the template provided.

Any generated interrupt service routines can be found at the end of the source file that the Code wizard saves to. It is recommended that you move them from the generated file and into a separate

Figure 6.8: Message panel of C-Wiz

file. The reason for this is that if you were to make modifications to this file, all changes will be lost if this file is regenerated at a later date.

6.9 Handling Shared Resources

If configuring several microcontroller peripherals, it is possible for select combinations of settings to cause two or more peripherals to rely upon access to the same resource simultaneously. If the device is allowed to operate in such a state, the devices can conflict it is possible that one or all the peripherals involved will not function as expected.

The Code wizard has the ability to identify hazardous conflicts between peripherals and will notify you whenever such a conflict occurs. In this event a brief report will appear in the code wizard's Messaging Panel. Figure 6.8 gives an example of a typical message generated when two peripherals are contesting for the same output pin on a microcontroller.

In this example the Messaging Panel indicates while the user was setting up the device's peripherals, four pin conflicts were identified. One of the reports from the messaging window is:

`PIN 2:A3 overrides P6.3`

This informs us that pin 2 of this microcontroller was initially set to `P6.3` (an I/O port pin mode). Later another peripheral also needed the use of pin2. The report explains that pin 2 switched from `P6.3` mode to `A3` mode (an analog channel). The identities of these modes: `A3`, `P6.3`, `CA0` etc. can be found in the manufacturer's data sheet for your selected microcontroller.

While a conflict is present, the Code wizard disables any settings and revokes any code that involves the disputed pin functioning in the defeated mode. In this example, any code used to set-up `P6.3` will be nullified while the settings that depend on `A3` still exist.

Resting the mouse over a disabled setting will produce a floating message identifying which

peripheral has created the conflict so that you know where to go in order to fix the problem. 6.5 shows an example of such a message. If the setting that created the conflict is reversed or adjusted in a way that avoids the conflict, a message such as this will be reported:

```
A3 surrenders PIN2
```

Note that if a pin function has been *surrendered*, this does not mean that it has been disabled. It simply means that in the current settings, that pin functionality is no longer required.

The handling of these shared resources (also known as dependency handling), can be toggled on and off using the **Advanced Options** dialog (see Subsection 6.2.6). Whilst the handling is disabled, all configurations are accepted and the code is generated accordingly. This means that two peripherals may be configured on the same pin and may operate unexpectedly. It is not recommended that the dependency handling be disabled unless the peripheral interactions are understood intimately.

6.10 Closing the Code Wizard

When finished with the Code wizard, there are two different types of closing procedures.

Firstly, closing via the **OK** button (see Figure 6.7): This type of exit will close the Code wizard and record all of the settings that have been made in the dialog and import a saved file to HI-TIDE if requested. The **OK** button is convenient if you plan to run the Code wizard again later in this project and don't want to have to start from scratch. Although closing via the **OK** button will record your peripheral settings, a generated source file will only contain the initialization code that existed at the time of the last file-save operation. This means that any changes made after the last save operation will not be present in the Code wizard's output file.

The alternative is to close via the **Cancel** button. Exiting this way will close the Code wizard without recording any new changes to the peripheral settings. If you were to re-run HI-TIDE again later and re-open the code wizard, it would only be restored to the point of the last file-save or **OK** operation. If neither of these events had ever occurred, the code wizard will not be able to restore any settings and will open in its initial blank state with all peripherals deactivated.

Closing the Code wizard via the exit button in the top-right corner of the dialog has the same effect as closing via the **Cancel** button.

Chapter 7

HI-TIDE Compiler Options

This chapter of the manual will explain each of the HI-TECH C compiler specific options that can be configured from within the **Global Compiler Options dialog** in HI-TIDE. To access the **Global** Compiler Options dialog select **Global Compiler Options...** from the **Project** menu.

Each compiler will have its own unique set of global compiler options which can be configured through the **Global Compiler Options dialog**. The global compiler options have been divided up into six sub-sections referred to by each tab in the **Global Compiler Options dialog**, these are:

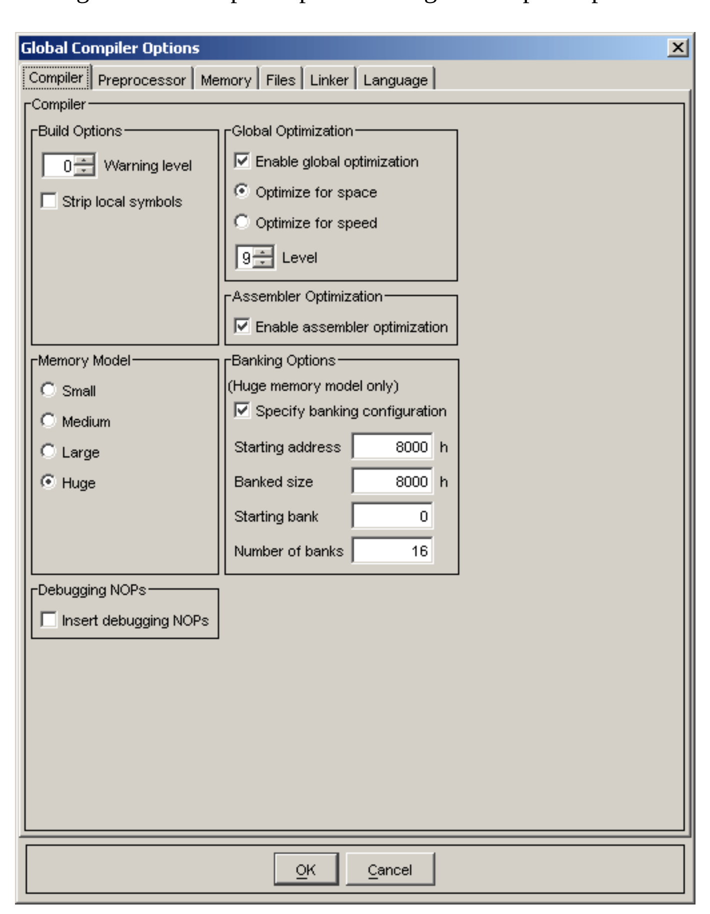
- Compiler options
- Preprocessor options
- Memory options
- Files options
- Linker options
- Language options

The purpose and behaviour of these options will be described briefly in the following sections, however you may want to refer to the HI-TECH C Compiler Options section (Section 10.4) for a more detailed explanation on how each option controls the compiler.

7.1 Compiler Options

Figure 7.1 shows the **Compiler Options** tab of the **Global Compiler Options dialog**. Each specific option is explained below.

Figure 7.1: Compiler options dialog — compiler options



7.1.1 Build options

7.1.1.1 Warning Level

Setting the **Warning level** value to anything other than zero enables the command-line driver `--WARN` option, (see Section 10.4.47) which is used to adjust the sensitivity level for compiler warning messages.

7.1.1.2 Strip Local Symbols

By

This option corresponds to the command-line driver `-X` option (see Section 10.4.17), which strips local symbols from any files compiled, assembled or linked.

7.1.2 Global Optimization

7.1.2.1 Enable Global Optimization

Selecting the **Enable global optimization** option will enable global code optimization. The level of optimization is specified in the **Level** option. Global optimization is also applied for speed or for space. For more information on global optimization, see Section 10.4.34.

7.1.2.2 Optimize For Speed / Space

Selecting the **Optimize for speed** or **Optimize for space** will respectively specify the suboptions `speed` or `space` to the `--OPT=` command-line driver option.

7.1.2.3 Level

The **Level** value will specific the level of optimization that will apply to the `--OPT=` command-line driver option (see Section 10.4.34). Note that **Enable global optimization** must be selected before the value of **Level** can be set.

7.1.3 Assembler Optimization

7.1.3.1 Enable Assembler Optimization

Selecting the **Enable assembler optimization** option will activate the assembler optimizer. This will set the command-line driver `--OPT` option to `--OPT=as_all`. For more information on assembler optimization, refer to Section 10.4.34.

Table 7.1: Memory model types

Memory model	**Setting**
small	s
medium	m
large	l
huge	h

7.1.4 Memory Model Settings

This option selects the memory model to implement. Figure 7.1 show the memory models available and the equivalent command-line driver option, `-B`, see Section 10.4.1.

7.1.5 Banking Options

The **Banking Options** setting is on enabled when the **Memory Model** option is set to **Huge**. This option is disabled if the **Memory Model** option is set to any other mode.

Selecting the **Specify banking configuration** option will enable the text fields to enter the banking configuration information. This will specify the `--BANK` option (see Section 10.4.19) on the command-line driver. The text fields will be added as the appropriate arguments.

7.1.6 Debugging NOPs

Selecting the Insert Debugging NOPs option will cause NOP instructions to be placed within the output code. These instructions are used to reserve space for certain debuggers. See Section 10.4.32 for more information.

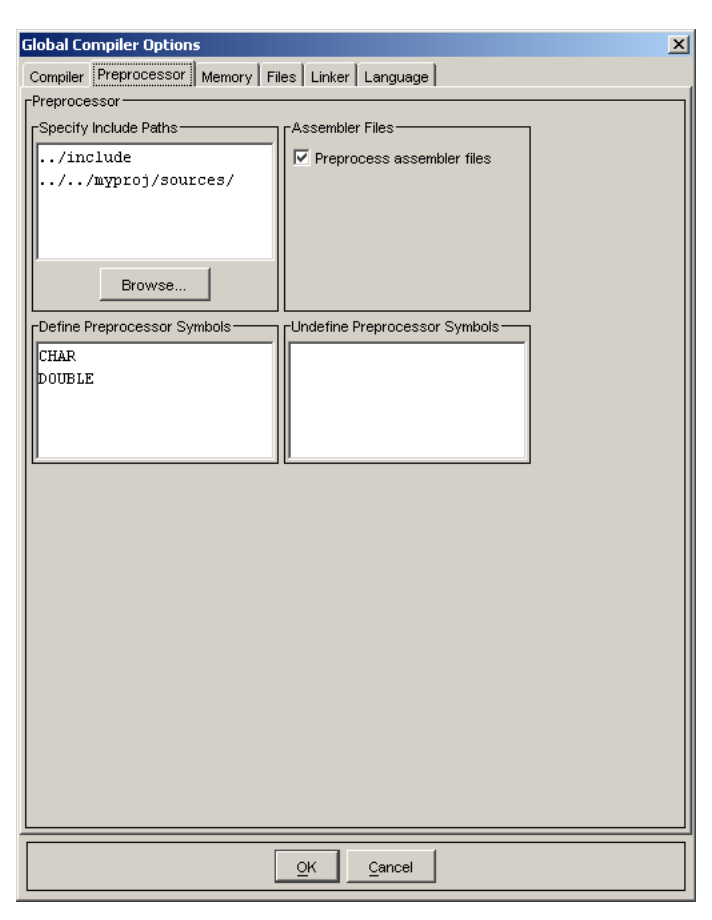
7.2 Preprocessor options

Figure 7.2 shows the **Preprocessor Options** tab of the **Global Compiler Options dialog**. Each preprocessor option is explained below.

7.2.1 Specify Include Paths

This is where you will specify the list of paths that the compiler will search in to find `#included` files. Include paths can be added by typing the include path in the list. Multiple include paths can be listed by entering each on a new line. Alternatively you can press the **Browse** button and select

Figure 7.2: Compiler options dialog — preprocessor options



a path from the **Browse** dialog. This option will specify the parameters for the `-I` command-line driver option (see Section 10.4.6).

7.2.2 Assembler Files

7.2.2.1 Preprocess assembler files

Selecting the **Preprocess assembler files** option will set the compiler to preprocess assembler files prior to assembling, thus allowing the use of preprocessor directives such as `#include`. This is equivalent setting `-P` on the command-line driver, see Section 10.4.12.

7.2.3 Define Preprocessor Symbols

This is where you will specify any symbols that you want to be defined and passed to the preprocessor. Multiple symbols can be defined if each appears on a new line. The values in the list will specify the parameters for the `-D` command-line driver option (see Section 10.4.3).

7.2.4 Undefine Preprocessor Symbols

This is where you will specify any symbols that will be passed to the preprocessor that are to be undefined during the preprocessing stage of the compile process. Symbols can be added to the list by typing the symbol name in the list and separating multiple symbols by a new line. The values in the list will specify the parameters for the `-U` command-line driver option (see Section 10.4.15)

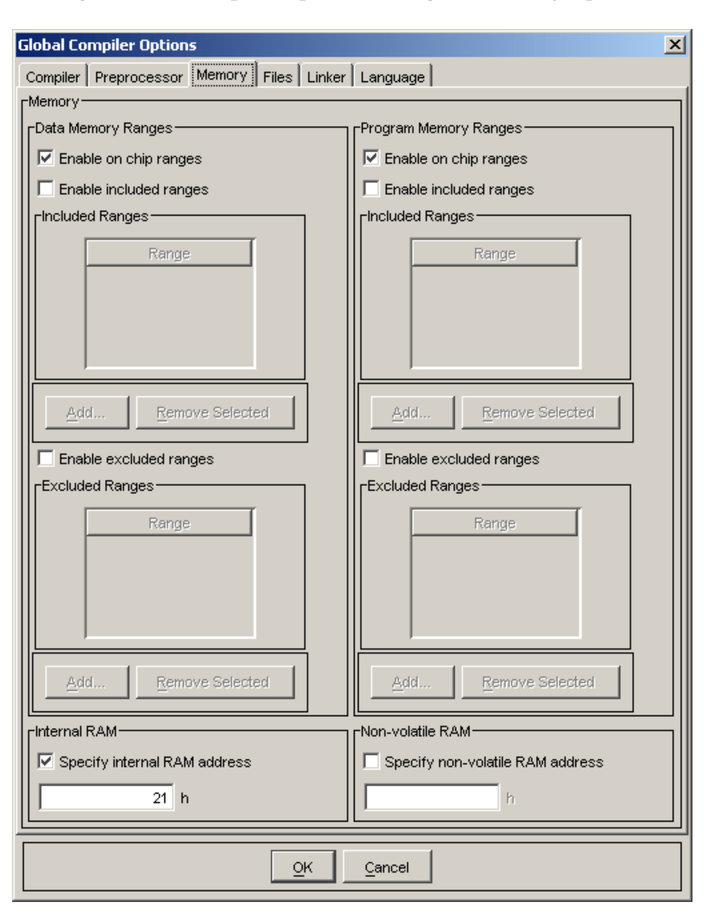
7.3 Memory options

Figure 7.3 shows the **Memory Options** tab of the **Global Compiler Options dialog**. Each specific option is explained below.

7.3.1 Program Memory Ranges

This area is where you can specify any additional program memory ranges or ROM that the compiler can use to store code in. If there are any areas of memory that you want to reserve from being used as program memory then you can specify these ranges in the *Excluded Ranges*. These options will specify the parameters for the `--ROM` command-line driver option (see Section 10.4.40).

Figure 7.3: Compiler options dialog — memory options



7.3.1.1 Enable on chip ranges

If the **Enable on chip ranges** checkbox is selected then the default memory ranges as specified in the chipinfo file will be available for the compiler to use as program memory. Otherwise these default memory ranges will not be used, unless specified in the *Included Ranges*.

7.3.1.2 Enable included ranges

If the **Enable included ranges** checkbox is selected then the memory ranges as specified in the *Included* Ranges list will be available for the compiler to use as program memory.

7.3.1.3 Included Ranges

This is where you can specify additional memory ranges that the compiler can use to store code in. To add a memory range you will need to press the **Add**... button, which will display the **Add Range** dialog, where you can specify the start address and the end address of the memory range. Once you have specified the start address and the end address, select **OK** to add the range to the list of *Included Ranges*. Note that you will not be able add memory ranges that overlap.

7.3.1.4 Enable excluded ranges

If the **Enable excluded ranges** checkbox is selected then the memory ranges as specified in *Excluded* Ranges list will be made unavailable for the compiler to use as program memory.

7.3.1.5 Excluded Ranges

This is where you can specify memory ranges that the compiler will exclude from storing code in. To add a memory range you will need to press the **Add**... button, which will display the **Add Range** dialog, where you can specify the start address and the end address of the memory range. Once you have specified the start address and the end address, select **OK** to add the range to the list of *Excluded Ranges*.

7.3.2 Data Memory Ranges

This area is where you can specify any additional data memory ranges or RAM that the compiler can use to store data in. If there are any areas of memory that you want to reserve from being used as data memory then you can specify these ranges in the *Excluded Ranges*. These options will specify the parameters for the `--RAM` command-line driver option (see Section 10.4.39).

7.3.2.1 Enable on chip ranges

If the **Enable on chip ranges** checkbox is selected then the default memory ranges as specified in the in the chipinfo file will be available for the compiler to use as data memory. Otherwise these default memory locations will not be used, unless specified in the *Included* Ranges.

7.3.2.2 Enable included ranges

If the **Enable included ranges** checkbox is selected then the memory ranges as specified in *Included* Ranges list will be available for the compiler to use as data memory.

7.3.2.3 Included Ranges

This is where you can specify additional memory ranges that the compiler can use to store data in. To add a memory range you will need to press the **Add**... button, which will display the **Add Range** dialog, where you can specify the start address and the end address of the memory range. Once you have specified the start address and the end address, select **OK** to add the range to the list of *Included Ranges*. Note that you will not be able add memory ranges that overlap.

7.3.2.4 Enable excluded ranges

If the **Enable excluded ranges** checkbox is selected then the memory ranges as specified in *Excluded* Ranges list will be made unavailable for the compiler to use as data memory.

7.3.2.5 Excluded Ranges

This is where you can specify memory ranges that the compiler will exclude from storing code in. To add a memory range you will need to press the **Add**... button, which will display the **Add Range** dialog, where you can specify the start address and the end address of the memory range. Once you have specified the start address and the end address, select **OK** to add the range to the list of *Excluded Ranges*. Note that you will not be able add memory ranges that overlap.

7.3.3 Internal RAM

Selecting the **Specify internal RAM address** option will enable the text field to enter the starting address of the internal RAM. This will specify the parameter for the `--INTRAM` command-line driver option (see Section 10.4.29).

7.3.4 Non-volatile RAM

Selecting the **Specify non-volatile RAM address** option will enable the text field to enter the starting address of the non-volatile RAM. This will specify the parameter for the `--NVRAM` command-line driver option (see Section 10.4.33).

7.4 Files options

Figure 7.4 shows the **File Options** tab of the **Global Compiler Options dialog**. Each specific option is explained below.

7.4.1 Output File Type

This options allows the type of the output file to be set. The path of the output file is shown in the text field. This option will specify the parameters for the `--OUTPUT` command-line driver option (see Section 10.4.36). The default output file type is grey out in the list of file types.

7.4.2 Debug Information

7.4.2.1 Generate assembler listing

If the **Generate assembler listing** checkbox is selected then an assembler listing file (.lst) will be generated for each C module in the project when compiled. This option will enable the `--ASMLIST` command-line driver option (see Section 10.4.18).

7.4.2.2 Generate map file

If the **Generate map file** checkbox is selected then a Map File will be generated for the project when compiled. This option will enable the `-M` command-line driver option (see Section 10.4.9).

7.5 Linker options

Figure 7.5 shows the **Linker Options** tab of the **Global Compiler Options dialog**. Each specific option is explained below.

Figure 7.4: Compiler options dialog — file options

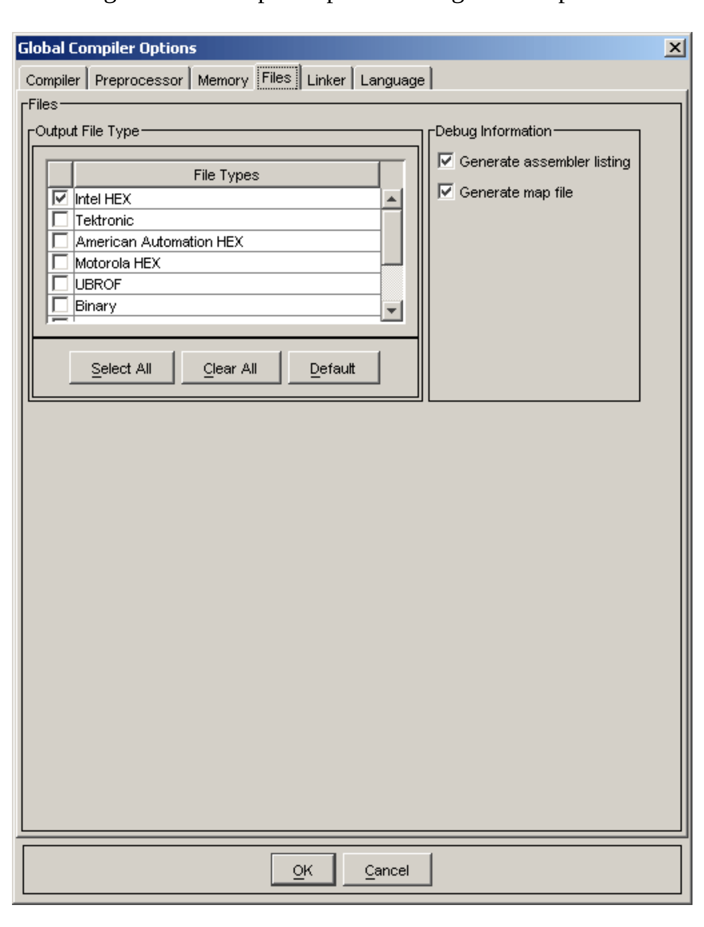
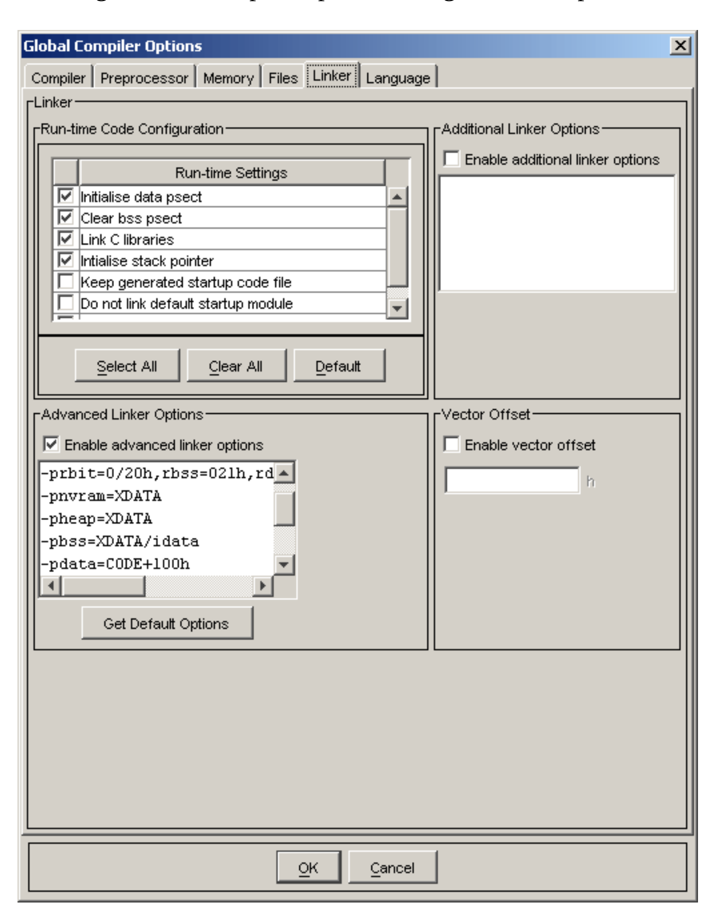


Figure 7.5: Compiler options dialog — linker options



7.5.1 Run-time Code Configuration

This section allows you to customize the Run-time code generated by the compiler. These options will specify the parameters for the `--RUNTIME` command-line driver option (see Section 10.4.41 for more information on all the available suboptions).

7.5.1.1 Run-time Settings

Initialize data psect Selecting the **Initialize data psect** checkbox will specify the `init` suboption to the `--RUNTIME` command-line driver option.

Clear bss psect Selecting the **Clear bss psect** checkbox will specify the `clear` suboption to the `--RUNTIME` command-line driver option.

Link C library Selecting the **Link C library** checkbox will specify the `c_libs` suboption to the `--RUNTIME` command-line driver option.

Initialize stack pointer Selecting the **Initialize stack pointer** checkbox will specify the `stack` suboption to the `--RUNTIME` command-line driver option.

Keep generated startup code file Selecting the **Keep generated startup code file** checkbox will remove the `keep` suboption from the --RUNTIME command-line driver option.

Do not link default startup module Selecting the **Do not link default startup module** checkbox will specify the `no_startup` suboption to the `--RUNTIME` command-line driver option.

Enable stack checking Selecting the **Enable stack checking** checkbox will specify the `stack_check` suboption to the --RUNTIME command-line driver option.

7.5.2 Vector Offset

Enable vector offset Selecting the **Enable vector offset** checkbox will allow you to enter the offset address. The address to offset can be entered in the text box. The address is in hexadecimal format (without the leading `0x`). This will specify the `--CODEOFFSET=`*`address`* option on the command-line driver (see Section 10.4.23).

7.5.3 Additional Linker Options

The *Additional Linker Options* section allows you to add additional Linker options.

7.5.3.1 Enable additional linker options

By selecting the **Enable additional linker options** you will be able to edit the list of additional linker options, that will be passed to the Linker.

7.5.4 Advanced Linker Options

The *Advanced Linker Options* section allows you to modify the default linker options. These options are maintained by HI-TIDE and should only be modified by advanced users of HI-TECH Software's C compilers.

7.5.4.1 Enable advanced linker options

By selecting the **Enable advanced linker options** checkbox you will be able to edit the list of advanced linker options, that will be passed to the Linker. By pressing the **Get Default Options** button the *Advanced Linker* options list will be updated with the default linker options.

7.6 Language options

Figure 7.6 shows the **Language Options** tab of the **Global Compiler Options dialog**. Each specific option is explained below.

7.6.1 Default Char Type

The **char type unsigned as default** will enable or disable all variables of type char to be unsigned or signed. This option will specify the parameters for the `--CHAR` command-line driver option (see Section 10.4.20).

7.6.2 Identifier Length

This option allows the C identifier length to be increased from the default value of 31. Valid sizes for this option are from 32 to 255. This option will specify the parameters for the `-N` command-line driver option (see Section 10.4.10).

7.6.3 ANSI Conformance

7.6.3.1 Enable strict ANSI conformance

This option will enable or disable strict ANSI compliance of all special keywords. This option will enable or disable the `--STRICT` command-line driver option (see Section 10.4.44).

Figure 7.6: Compiler options dialog — language options

Chapter 8

HI-TIDE Compilation

HI-TIDE provides a graphical, user-friendly way of running HI-TECH Software's C compilers. Options are set through dialog boxes which allow for ease-of-use, but still provide access to the advanced features of the compiler, such as specifying custom linker options. Dependency checking is also handled by the IDE by only compiling the files that need to be recompiled, thus decreasing compilation time. Errors or warnings in the source code are reported in an easy-to-read format which also allows the location of the error to be highlighted in the built-in editor.

8.1 Compiling Project Files

Building a project is broken into two main steps: compilation and linking, which are both detailed below. Each step is accompanied by a description of what is involved in each step and a simple example. The project must have at least one source or object file in the project to be able to be compiled.

8.1.1 Compiling Source Files

Step one is compiling the C and/or assembler source files in the project. Compiling a source file is the act of running the compiler with either the C or assembler file as an input file to produce a relocatable object file. For brevity, C and assembler files for the rest of this chapter will be referred to as *source files*. The compiler is run by HI-TIDE for each source file in the project. The relocatable object file produced will be placed into the project directory. Object and library files are not handled in this step as they are both in a relocatable object file format, or a variation of this in the case of library files.

Errors and warnings are issued by the compiler when the compiler detects an error or warning in the files that are being compiled. If errors occur when compiling a source file, the compiler will stop compiling the source file and, by default, compile the next source file. An option to stop compiling source files if errors occur is available by deselecting the **Stop on Errors** option on the **Project** tab of the **General Preferences** dialog.

•

For example, a project called `example1` contains the files `main.c`, `math.as`, `common.obj` and `lcd.lib`. To compile the source files the compiler will be run to compile `main.c` to the relocatable object file `main.obj`. The compiler will be run again to compile `math.as` to the relocatable object file `math.obj`. The files `common.obj` and `lcd.lib` are not used in this step.

8.1.2 Linking

Step two is linking. Linking is the act of running the compiler, with one or more relocatable object and/or library files as input files, to produce one or more output files (for example, Intel HEX file). The input files are the relocatable object files produced in the first step of compiling and any of the object and/or library files specified in the project. The output file produced will be placed into the project directory. The link step is run at most once when compiling.

•

Following on from the example in the previous step, to link, the compiler will be run with `main.obj`, `math.obj`, `common.obj` and `lcd.lib` as input files and will produce the output file `example1.hex`.

8.1.3 Make

Make will compile the project files performing dependency checking, so that only source files that are out-of-date are recompiled and linking is only performed when necessary. A list of conditions which cause files to be recompiled and/or a link to be performed are show in 8.1. To invoke make, select **Make** from the **Build** menu or the **Make** toolbar button.

Before a make begins the project source files that are being edited will be checked to see if they have been modified. The action that is performed when a source file has been modified is specified by the **Save Modified File Before Building** option on the **Editor** tab of the **General Preferences**

Table 8.1: Recompile Conditions

Cause	Effect
global compiler options changed	all source files will be recompiled and the program relinked
the file-specific compiler options changed for one or more files	source file(s) whose options have changed will be recompiled and the program relinked
one or more source files have changed or have been added to the project	the changed or new source file(s) will be recompiled and the program relinked
a source file's dependency has changed	the source file will be recompiled and the program relinked
object file or library file has changed	the program will be relinked
one or more files have been removed from the project	the program will be relinked

dialog. There are three options available, **Always** save modified file, **Never** save modified file and **Prompt** to save modified file. By default this option is set to always save modified file.

The always save modified file option automatically saves the file before building. There isn't a prompt shown and the file is saved to disk. This means that when compilation begins, the most up-to-date version of the file will be compiled.

The never save modified file option will not save the modified file. This means that when compilation begins, the contents of the file saved to disk — not the modified file in the editor — is used during compilation.

The prompt to save modified file option will show a dialog before make begins for each modified project source file in the editor. The dialog will prompt to save the modified file. Selecting Yes will save the file to disk. Selecting No will leave the file on disk unmodified.

If no errors are issued during a make it is considered a successful make. This also means that the compiler was able to produce an updated HEX file. Warnings do not effect if the compilation was successful but they may be important when debugging the application in development. After a successful compilation, by default, HI-TIDE will automatically load the HEX file produced. This functionality can be disabled by deselecting the **Load HEX File on Successful Build** option on the **Project** tab of the **General Preferences** dialog.

8.1.4 Make All

Make all, as the name suggests, will compile and link all files in the project. To perform this operation, select **Make All** from the **Build** menu. A check to ensure that each project source file open in the editor is unmodified is preformed as per the Make operation, described above. The resultant HEX file is loaded into the debugger after a successful make as described in the Make operation above.

8.1.5 Individual Files

Each source file is able to be compiled to a number of intermediate files. Intermediate files are various file types that can be produced from the compiler when compiling an individual file. The intermediate files that can be produced for a C source file is assembler files, function prototype files, object files and preprocessed files. The intermediate files that can be produced for an assembler source file are preprocessed files and object files. The intermediate files produced will be placed into the project directory.

To compile a source file to an intermediate file right click on a source file icon in the Project view and select the **Compile To** sub menu. From the sub menu select the intermediate file to compile to. If a project source file is open in the editor it can be compiled to an object file by selecting **Compile To** Object File from the **Build** menu.

8.2 Compiler Options

Compiler options specify settings that will be passed to the compiler when the compiler is run. The **Global Compiler Options** dialog is a common place where the options used for compiling and linking are specified. By default all source files use the options specified in the **Global Compiler Options** dialog when they are compiled. If the options for an individual file are required to be different to that of the other files then a set of *file-specific* options may be specified for each file.

8.2.1 Global Compiler Options

Global options that apply to all files that do not have file-specific options set. These options are always used in the link step. To open the **Global Compiler Options** dialog select **Global Compiler Options...** menu from the **Project** menu or double click on the output file icon in the Project view.

8.2.2 File-Specific Compiler Options

To enable file-specific compiler options for a particular file, right click on the source file icon in the Project view and select **File Specific Compiler Options...** from the popup menu. This will open a

dialog that is similar to the **Global Compiler Options** dialog. When this dialog is initially opened, the state of the options will be the same as the current global options, however once they have been enabled they will hold their state.

This contains the same options found in the global options dialog, only some options will be disabled. These are options that do not apply to the file, for example, preprocess assembler file options is disabled when setting file-specific options for a C file. Initially all of the available options are disabled. To specify and use the file-specific options they need to be enabled by selecting **Use File-specific Options** checkbox which is present at the top of the dialog. This check box is always visible regardless of which tab is selected and applies to all the tabs in the dialog.

Those options required may be selected in the usual way, however if an option is changed from the corresponding option specified in the global compiler options, then the widget will change colour to red. This allows the options that differ to the global compiler options to be easily identified.

All options under all tabs may be returned to those specified by the **Global Compiler Options** dialog by selecting the **Revert to Global Compiler Options** button.

8.3 Build Results

Build results displays messages issued by the compiler after compilation is finished. Finished compilation is defined as the end of a make, make all or an individual file is compiled. The messages are organised by the Build view**,** discussed in Section 4.2. These messages are broken up into four categories, errors and warnings, memory usage, psect usage and build log. The categories are discussed below.

8.3.1 Error and Warnings

Errors and warnings are issued by the compiler when the compiler detects an error or warning in the files that are being compiled. The error or warning contains information on where the problem occurred and a short description of what the error or warning means. The errors and/or warnings, if any, are detected by HI-TIDE and displayed in the **Error Log** tab of the Build view. Double-clicking on an error or warning in the error log will, for most cases, open the file that caused the error in the editor and place the caret on the line the error occurred. See Section 4.2.1 for more details on how there errors are displayed and how the view is accessed.

8.3.2 Memory Usage

Memory usage displays memory information of the application being developed. This information is only updated after successful completion of a **Make** or **Make All**. It is not updated after an individual file is compiled. It is displayed in the **Memory Usage** tab of the Build view.

8.3.3 Psect Usage

Psect usage displays memory usage of program sections of the application being developed. This information is only updated after successful completion of a **Make** or **Make All**. It is not updated after an individual file is compiled. It is displayed in the **Psect Usage** tab of the Build view.

8.3.4 Build Log

The build log displays detailed information on the commands used to run the compiler, messages returned from the compiler, details on dependency checking, the type of options that are being used and build times. The build log displayed in the **Build Log** tab of the Build view. See Section 4.2.4 for more details on how the view is accessed.

Each time the compiler is run, the command line that is used to run the compiler is displayed in the build log. The command line will then be followed by a verbose output of the compiler.

Errors and warnings are displayed in the build log as well as the error log. The errors in the build log are displayed in the format output by the compiler and are indicated by the text **(error)** at the end of a line. A warning will be indicated by the text **(warning)** at the end of a line. The errors and warnings will appear after the verbose output of the compiler.

When performing a **Make**, dependency checking is performed and a description of the checking process is shown in the build log. The description indicates if the file is up-to-date or, if it is not up-to-date, why it is not up-to-date. The description appears before the compiler command line.

Just before the compiler command line a text description of the whether file-specific or global compiler options are used to compile the file is displayed. The message `Using global options...` indicates global options are being used to compile the source file. The message `Using file specific options...` indicates file-specific options are being used to compile the file.

When a **Make** is performed the log is titled with the text *Making Project* that indicates a make with dependency checking is performed. For a **Make All** the log is titled with the text *Making All Project* Files. Following the title for both the **Make** and the **Make All** is a message that indicates the date and the time that the build was started.

Chapter 9

HI-TIDE Debugging

HI-TIDE provides a generic debugger interface which allows integration of a wide range of simulators and emulators available for a particular chipset. At present HI-TIDE can be used to load HEX files, manage breakpoints, perform C and assembly stepping, as well as animate (multi step) code execution while watching program variables, registers, and memory.

9.1 Debugger Functions

The following sections provide an overview of the debugger's functions available in HI-TIDE. More specific information on debugger views and related user interface aspects may be found in 4.4. Related toolbar actions and buttons are described in 3.2.8.

9.1.1 Debugger Initialization

Before debugging can be performed, a HEX file has to be loaded into the device, whether the device is a simulator, emulator or an actual microcontroller. In order to allow high-level debugging features, such as source-level breakpoints and variable display, the compiler must have produced the appropriate symbolic debug information. Typically this happens by default, but compiler options can control this aspect of the compiler. See 10.4.5 for more information.

Once the HEX file is successfully loaded, the debug menus and buttons are enabled and further debugging may be undertaken.

9.1.2 Breakpoints

A breakpoint is a point in a program that, when reached, triggers some special behaviour. Generally, breakpoints are used to pause program execution and allow the values of some or all of the program variables to be inspected. Breakpoints may consist of instructions that form part of the program or they may be temporarily set by the programmer through HI-TIDE.

In HI-TIDE a breakpoint can be set anywhere within the executable memory range. Breakpoints can be set from a Disassembly view, see Section 4.4.1.4, or from a source file open in the Editor view, see Section 4.3.9. A red breakpoint dot in the view gutter indicates that the breakpoint is set. Once set, breakpoints can either be removed completely or disabled so that execution will no longer stop at that location. Disabled breakpoints are indicated by a grey breakpoint dot in the view's gutter.

Whilst running, animating or single stepping, if the program execution point reaches an enabled breakpoint, the debugger will stop immediately without executing the C statement, or assembler instruction, at which the breakpoint is set. That is, the next assembler of C statement to be executed will be that marked with the red breakpoint dot.

A single C statement may correspond to several assembly instructions. When a breakpoint is set on a C statement, the breakpoint is essentially set on the first assembly instruction that corresponds to that statement. As well as the breakpoint indicator in the Editor view, another red breakpoint dot will appear in the Disassembly view at the first assembly instruction that corresponds to the line of C code on which the breakpoint was set. These two indicators represent the same breakpoint. Disabling one will disable the other, and similarly removing one will remove both. The reverse is also true: setting a breakpoint on the first assembly instruction associated with a C statement will also set a breakpoint indicator in the Editor view.

9.1.2.1 Breakpoint Restoration

Not all breakpoints are preserved by HI-TIDE. After compilation some breakpoints are removed, and when restoring a project file, some breakpoints that may have been set when the project was saved will no longer be set. The breakpoints affected are those set on assembly instructions that are not the first instruction associated with a C statement and those set on assembly instructions which do not immediate follow a global assembly label.

These breakpoints are removed as there is no means available to track the position of these instructions as the program is changed. All breakpoints set in the Editor view will be fully preserved.

Even if a breakpoint is preserved, its position in the program may change. As a program is edited, the position of breakpoints will remain at the assembly address at which they were set. Breakpoints set in the Editor view will also remain positioned at the assembly address which corresponds to the C statement at which the breakpoint was set. If the contents of the Editor view change, the breakpoint may map back to a different C statement to which it was originally assigned.

9.1.3 Program execution

HI-TIDE provides several modes of program execution that are useful during the debugging process. The important differences between modes are listed below.

9.1.3.1 Run

When Run mode is selected, the debugger executes code in real time (or near real time in case of simulator). This mode is resource intensive for the microcontroller which prevents real-time access to microcontroller registers and memory. Execution continues until stopped by the user or a breakpoint is encountered.

No views are updated during Run mode, but will update as soon as the program is stopped.

9.1.3.2 Animate

When Animate mode is selected, the debugger executes one assembly instruction and then updates any debugger view visible in the Workspace area. If the Editor view is visible, it is also updated to indicate the current program position. Execution continues until stopped by the user or a breakpoint is encountered.

Unlike Run mode, Animate does not allow for a real-time execution and so may be unsuitable for time-critical programs, such as USART communication routines. This mode might be somewhat slow when used with hardware emulators due to the amount of time required to acquire memory and register information from the device, however Animate mode is faster and more convenient that repeatedly single stepping through assembly code.

9.1.3.3 Assembly Step

Assembly Step mode causes debugger to execute a single assembly instruction then stop. All views are updated after the step.

9.1.3.4 C Step

C step mode causes the debugger to execute a series of assembly steps that correspond to a single line of C code. This mode is similar to Animate mode, but no views are updated after each step, and execution will stop when the first assembly instruction corresponding to the next C statement is encountered. C step mode does not perform real-time execution.

A compound C statement, e.g. a `for()` or `while()` loop, is executed as one C statement. C stepping a compound statement will run all iterations of the loop or the sub statements of the `if()` etc.

C stepping a C function call will step into the function and execution will stop at the first C statement in the function being called. If the function is a library function, or any function for which there is no symbolic debug information, the entire C function called is run and execution will stop at the C statement immediately following the function call. The same is also true if the function called is written in assembly code.

9.1.3.5 Reset

Selecting reset either causes a hardware reset in the emulator or development board or a simulated reset if the selected debugger is the simulator. The exact nature of a hardware reset depends on the attached hardware. As a general rule, the program counter is set to starting location as specified by the reset vector.

9.2 8051 Debuggers

HI-TIDE supports debugging of 8051 family microcontrollers through use of a 8051 simulator.

9.2.1 Simulator

The 8051 simulator is a software debugger that can be used with all chips supported in the HI-TECH C for 8051 package. Code for chips which support memory banking cannot be simulated when compiled with the huge (banked) memory model, but can be used when compiled with other memory models.

The simulator provides the microcontroller, memory and SFR simulation.

The 8051 simulator poses no limitations, excluding those implicit to all simulators. The user has to keep in mind that the simulator is only an approximation of the physical device and that a hardware testing is generally required to ensure correctness of the implemented solution.

Chapter 10

C51 Command-line Driver

C51 is the driver invoked from the command line to compile and/or link C programs. `C51` has the following basic command format:

```
C51 [options] files [libraries]
```

It is conventional to supply the options (identified by a leading *dash* "-" or *double dash* "–") before the filenames.

The options are discussed below. The files may be a mixture of source files (C or assembler) and object files. The order of the files is not important, except that it will affect the order in which code or data appears in memory. *Libraries* are a list of library names, or `-L` options. See Section 10.4.7. Source files, object files and library files are distinguished by `C51` solely by the *file type* or *extension*. Recognized file types are listed in Table 10.1. This means, for example, that an assembler file must always have a `"."` `as` extension (alphabetic case is not important).

C51 will check each file argument and perform appropriate actions. C files will be compiled; assembler files will be assembled. At the end, unless suppressed by one of the options discussed later,

Table 10.1: C51 file types

File Type	Meaning
`.c`	C source file
`.as`	Assembler source file
`.obj`	Relocatable object code file
`.lib`	Relocatable object library file

all object files resulting from compilation or assembly, or those listed explicitly on the command line, will be linked together with the standard runtime code and libraries and any user-specified libraries. Functions in libraries will be linked into the resulting output file only if referenced in the source code.

Invoking `C51` with only object files specified as the file arguments (i.e. no source files) will mean only the link stage is performed. It is typical in Makefiles to use `C51` with a `-C` option to compile several source files to object files, then to create the final program by invoking `C51` again with only the generated object files and appropriate libraries (and appropriate options).

10.1 Long Command Lines

The `C51` driver is capable of processing command lines exceeding any operating system limitation. To do this, the driver may be passed options via a command file. The command file is read by using the `@` symbol. For example:

```
C51 @xyz.cmd
```

10.2 Default Libraries

C51 will search the appropriate standard C library by default for symbol definitions. This will always be done last, after any user-specified libraries. The particular library used will be dependent on the processor selected.

10.3 Standard Runtime Code

C51 will also automatically generate standard runtime start-up code appropriate for the processor and options selected unless you have specified the to disable this via the `--RUNTIME` option. If you require any special powerup initialization, you should use the *powerup* routine feature (see Section 11.1.5).

10.4 C51 Compiler Options

Most aspects of the compilation can be controlled using the command-line driver, `C51`. The driver will configure and execute all required applications, such as the code generator, assembler and linker.

C51 recognizes the compiler options listed in Table 10.2. The case of the options is not important, however UNIX shells are case sensitive when it comes to names of files.

Table 10.2 C51 command-line options

Option	Meaning
-B*model*	Specify memory model
-C	Compile to object file only
-D*macro*	Define preprocessor macro
-E+*file*	Redirect and optionally append errors to a file
-G*file*	Generate source-level debugging information
-I*path*	Specify a directory pathname for include files
-L*library*	Specify a library to be scanned by the linker
-L-*option*	Specify *-option* to be passed directly to the linker
-M*file*	Request generation of a MAP file
-N*size*	Specify identifier length
-O*file*	Output file name
-P	Preprocess assembler files
-Q	Specify quiet mode
-S	Compile to assembler source files only
-U*symbol*	Undefine a predefined preprocessor symbol
-V	Verbose: display compiler pass command lines
-X	Eliminate local symbols from symbol table
--ASMLIST	Generate assembler .LST file for each compilation
--BANK=*argument*	Specify banking options
--CHAR=*type*	Make the default char signed or unsigned
--CHIP=*processor*	Selects which processor to compile for
--CHIPINFO	Displays a list of supported processors
--CODEOFFSET=address	Specify a code offset
--CR=*file*	Generate cross-reference listing
--ERRFORMAT<=*format*>	Format error message strings to the given style
--GETOPTION=*app,file*	Get the command line options for the named application
--HELP<=option>	Display the compiler's command line options
--IDE=*ide*	Configure the compiler for use by the named IDE
--INTRAM=*address*	Specify internal RAM address
--MEMMAP=*file*	Display memory summary information for the map file
--NOEXEC	Go through the motions of compiling without actually compiling
--NOPS	Insert debug NOPs
--NVRAM=*address*	Specify non-volatile RAM address
continued...	

Option	Meaning
`--OPT<=`*`type`*`>`	Enable compiler optimizations
`--OUTDIR=`*`directory`*	Specify output directory
`--OUTPUT=`*`type`*	Generate output file type
`--PRE`	Produce preprocessed source files
`--PROTO`	Generate function prototype information
`--RAM=lo-hi<,lo-hi,...>`	Specify and/or reserve RAM ranges
`--ROM=lo-hi<,lo-hi,...>|tag`	Specify and/or reserve ROM ranges
`--RUNTIME=`*`type`*	Configure the C runtime libraries to the specified type
`--SCANDEP`	Generate file dependency ".DEP files"
`--SETOPTION=`*`app,file`*	Set the command line options for the named application
`--SETUP=argument`	Setup the product
`--STRICT`	Enable strict ANSI keyword conformance
`--SUMMARY=`*`type`*	Selects the type of memory summary output
`--VER`	Display the compiler's version number
`--WARN=`*`level`*	Set the compiler's warning level

All single letter options are identified by a leading *dash* character, `-`, e.g. `-C`. Some single letter options specify an additional data field which follows the option name immediately and without any whitespace, e.g. `-Ddebug`.

Multi-letter, or word, options have two leading *dash* characters, e.g. `--ASMLIST`. (Because of the double *dash*, you can determine that the option `--ASMLIST`, for example, is not a `-A` option followed by the argument `SMLIST`.) Some of these options define suboptions which typically appear as a *comma*-separated list following an *equal* character, `=`, e.g. `--OUTPUT=hex,omf`. The exact format of the options varies and are described in detail in the following sections.

Some commonly used suboptions include `default`, which represent the default specification that would be used if this option was absent altogether; `all`, which indicates that all the available suboptions should be enabled as if they had each been listed; and `none`, which indicates that all suboptions should be disabled. Some suboptions may be prefixed with a plus character, `+`, to indicate that they are in addition to the other suboptions present, or a minus character, `-`, to indicate that they should be excluded. In the following sections, *angle brackets*, < >, are used to indicate optional parts of the command.

10.4.1 -B: Specify Memory Model

The `-B` option is used to select the type of code generation desired according to the requirements of your program. The available memory models are shown in Table 10.3. See section 11.6 for more details about the various memory models available, and section 11.15 for further tips on choosing a memory model which suits your program.

Table 10.3: Memory model options

Option	Memory Model
`-Bs`	Small
`-Bm`	Medium
`-Bl`	Large
`-Bh`	Huge

See Section 7.1.4 for more information on how to specify the memory model from within HI-TIDE.

10.4.2 `-C`: Compile to Object File

The `-C` option is used to halt compilation after generating a relocatable object file. This option is frequently used when compiling multiple source files using a "make" utility. If multiple source files are specified to the compiler each will be compiled to a separate `.obj` file. The object files will be placed in the directory in which `C51` was invoked, to handle situations where source files are located in read-only directories. To compile three source files `main.c`, `module1.c` and `asmcode.as` to object files you could use a command similar to:

```
C51 --CHIP=8051AH -C main.c module1.c asmcode.as
```

The compiler will produce three object files `main.obj`, `module1.obj` and `asmcode.obj` which could then be linked to produce an *Intel* HEX file using the command:

```
C51 --CHIP=8051AH main.obj module1.obj asmcode.obj
```

See Sections 4.1.1.3 and 4.1.1.5for more information on how to compile to object files from within HI-TIDE.

10.4.3 `-D`*`macro`*: Define Macro

The `-D` option is used to define a preprocessor macro on the command line, exactly as if it had been defined using a `#define` directive in the source code. This option may take one of two forms, `-Dmacro` which is equivalent to:

```
#define macro 1
```

placed at the top of each module compiled using this option, or `-Dmacro=`*text* which is equivalent to:

```
#define  macro  text
```

where ***text*** is the textual substitution required. Thus, the command:

```
C51 --CHIP=8051AH -Ddebug -Dbuffers=10 test.c
```

will compile `test.c` with macros defined exactly as if the C source code had included the directives:

```
#define  debug    1
#define  buffers 10
```

See Section 7.2.3 for information on how to define macros when compiling within HI-TIDE.

10.4.4 -E*file*: Redirect Compiler Errors to a File

Some editors do not allow the standard command line redirection facilities to be used when invoking the compiler. To work with these editors, `C51` allows an error listing filename to be specified as part of the `-E` option. Error files generated using this option will always be in `-E` format. For example, to compile `x.c` and redirect all errors to `x.err`, use the command:

```
C51 --CHIP=8051AH -Ex.err x.c
```

The `-E` option also allows errors to be appended to an existing file by specifying an *addition* character, `+`, at the start of the error filename, for example:

```
C51 --CHIP=8051AH -E+x.err y.c
```

If you wish to compile several files and combine all of the errors generated into a single text file, use the `-E` option to create the file then use `-E+` when compiling all the other source files. For example, to compile a number of files with all errors combined into a file called `project.err`, you could use the -E option as follows:

```
C51 --CHIP=8051AH -Eproject.err -O -Zg -C main.c
C51 --CHIP=8051AH -E+project.err -O -Zg -C part1.c
C51 --CHIP=8051AH -E+project.err -C asmcode.as
```

The file `project.err` will contain any errors from `main.c`, followed by the errors from `part1.c` and then `asmcode.as`, for example:

```
main.c 11 22: ) expected
main.c 63 0: ; expected
part1.c 5 0: type redeclared
part1.c 5 0: argument list conflicts with prototype
asmcode.as 14 0: Syntax error
asmcode.as 355 0: Undefined symbol _putint
```

10.4.5 **-G*file***: Generate source-level Symbol File

The -G option generates a *source-level symbol file* (i.e. a file which allows tools to determine which line of source code is associated with machine code instructions, and determine which source-level variable names correspond with areas of memory, etc.) for use with HI-TECH Software debuggers and simulators such as *HI-TIDE*. If no filename is given, the symbol file will have the same base name as the first source or object file specified on the command line, and an extension of .sym. For example the option -GTEST.SYM generates a symbol file called test.sym. Symbol files generated using the -G option include source-level information for use with source-level debuggers.

Note that all source files for which source-level debugging is required should be compiled with the -G option. The option is also required at the link stage, if this is performed separately. For example:

```
C51 --CHIP=8051AH -G -C test.c
C51 --CHIP=8051AH -C module1.c
C51 --CHIP=8051AH -Gtest.sym test.obj module1.obj
```

will include source-level debugging information for test.c only because module1.c was not compiled with the -G option.

The --IDE option will typically enable the -G option.

This option will also enable source-level debug information for assembler source files, see Section 12.2 for the assembler's -v option. Source-level debug information for the runtime startup module will also be enabled if the startup module is not deleted. See Section 11.1.4 for information on how to preserve the startup module.

10.4.6 **-I*path***: Include Search Path

Use -I to specify an additional directory to use when searching for header files which have been included using the #include directive. The -I option can be used more than once if multiple directories are to be searched. The default include directory containing all standard header files are always searched even if no -I option is present. The default search path is searched after any user-specified directories have been searched. For example:

```
C51 --CHIP=8051AH -C -Ic:\include -Id:\myapp\include test.c
```

will search the directories `c:\include` and `d:\myapp\include` for any header files included into the source code, then search the default compiler include directory which is typically located at `c:\htsoft\8051-c_<version>\include`.

See Section 7.2.1 for information on how to specify include paths when compiling within HI-TIDE.

10.4.7 -L*library*: Scan Library

The `-L` option is used to specify additional libraries which are to be scanned by the linker. Libraries specified using the `-L` option are scanned before the standard C library, allowing additional versions of standard library functions to be accessed.

The argument to -L is a library keyword to which the prefix `51` and the suffix `.lib` are added. Thus the option `-Lmylib` will, for example, scan the library `51mylib.lib` and the option `-Lxx` will scan a library called `51xx.lib`. All libraries must be located in the LIB subdirectory of the compiler installation directory. As indicated, the argument to the `-L` option is *not* a complete library filename.

If you wish the linker to scan libraries whose names do not follow the above naming convention or whose locations are not in the LIB subdirectory, simply include the libraries' names on the command line along with your source files. Alternatively, the linker may be invoked directly allowing the user to manually specify all the libraries to be scanned.

10.4.8 -L-*option*: Adjust Linker Options Directly

The `-L` option can also be used to specify an extra "-" option which will be passed directly to the linker by `C51`. If `-L` is followed immediately by any text starting with a *dash* character "-", the text will be passed directly to the linker without being interpreted by `C51`. For example, if the option `-L-FOO` is specified, the `-FOO` option will be passed on to the linker when it is invoked.

The `-L` option is especially useful when linking code which contains extra program sections (or *psects*), as may be the case if the program contains C code which makes use of the `#pragma psect` directive or assembler code which contains user-defined psects. See Section 11.12.3 for more information. If this `-L` option did not exist, it would be necessary to invoke the linker manually to link code which uses the extra psects.

One commonly used linker option is `-N`, which sorts the symbol table in the map file by address, rather than by name. This would be passed to `C51` as the option `-L-N`.

The `-L` option can also be used to replace default linker options. If the string starting from the first character after the -L up to the = character matches a default option, then the default option is replaced by the option specified. For example, `-L-pstack=2000h` will inform the linker to replace the default option that places the `stack` psect to be one that places the psect at the address 2000h. The default option that you are replacing must contain an *equal* character.

See Sections 7.5.3 and 7.5.4 for information on how to define additional, and modify existing, linker options when compiling within HI-TIDE.

10.4.9 `-Mfile`: Generate Map File

The `-M` option is used to request the generation of a map file. The map is generated by the linker an includes information about where objects are located in memory. If no filename is specified, then the name of the map file will have the same name as the first file listed on the command line, with the extension `.map`.

See Section 7.4.2.2 for information on how to create map files when compiling within HI-TIDE.

10.4.10 `-Nsize`: Identifier Length

This option allows the C identifier length to be increased from the default value of 31. Valid sizes for this option are from 32 to 255. The option has no effect for all other values.

See Section 7.6.2 for information on how to specify identifier length when compiling within HI-TIDE.

10.4.11 `-Ofile`: Specify Output File

This option allows the name of the output file(s) to be specified. If no `-O` option is given, the output file(s) will be named after the first source or object file on the command line. The files controlled are any produced by the linker or applications run subsequent to that, e.g. `CROMWELL`. So for instance the HEX file, map file and SYM file are all controlled by the `-O` option.

The `-O` option can also change the directory in which the output file is located by include the required path before the filename, e.g. `-Oc:\project\output\first.hex`. This will then also specify the output directory for any files produced by the linker or subsequently run applications.

10.4.12 `-P`: Preprocess Assembly Files

The `-P` option causes the assembler files to be preprocessed before they are assembled thus allowing the use of preprocessor directives, such as `#include`, with assembler code. By default, assembler files are not preprocessed.

See Section 7.2.2.1 for information on how to preprocess assembler files when compiling within HI-TIDE.

10.4.13 -Q: Quiet Mode

This option places the compiler in a *quiet mode* which suppresses the HI-TECH Software copyright notice from being displayed.

10.4.14 -S: Compile to Assembler Code

The -S option stops compilation after generating an assembler source file. An assembler file will be generated for each C source file passed on the command line. The command:

```
C51 --CHIP=8051AH -S test.c
```

will produce an assembler file called test.as which contains the code generated from test.c. This option is particularly useful for checking function calling conventions and signature values when attempting to write external assembly language routines. The file produced by this option differs to that produced by the --ASMLIST option in that it does not contain op-codes or addresses and it may be used as a source file and subsequently passed to the assembler to be assembled.

See Section 4.1.1.3 for information on how to compile to assembler files when compiling within HI-TIDE.

10.4.15 -U*macro*: Undefine a Macro

The -U option, the inverse of the -D option, is used to *undefine* predefined macros. This option takes the form -Umacro. The option, -Udraft, for example, is equivalent to:

```
#undef  draft
```

placed at the top of each module compiled using this option.

See Section 7.2.4 for information on how to undefine macros when compiling within HI-TIDE.

10.4.16 -V: Verbose Compile

The -V is the *verbose* option. The compiler will display the full command lines used to invoke each of the compiler applications or compiler passes. This option may be useful for determining the exact linker options if you need to directly invoke the HLINK command.

10.4.17 -X: Strip Local Symbols

The option -X strips local symbols from any files compiled, assembled or linked. Only global symbols will remain in any object files or symbol files produced.

See Section 7.1.1.2 for information on how to strip local symbols when compiling within HI-TIDE.

10.4.18 --ASMLIST: Generate Assembler .LST Files

The `--ASMLIST` option tells `C51` to generate an *assembler listing file* for each module being compiled. The list file shows both the original C code, and the generated assembler code and the corresponding binary op-codes. The listing file will have the same name as the source file, and a file type (extension) of `.lst`. The listing file will be relative — instructions will be shown at an address offset within their psect.

See Section 7.4.2.1 for information on how to create assembler listing files when compiling within HI-TIDE.

10.4.19 --BANK: Specify Banking Options

The `--BANK` option tells C51 the banking information to use when using huge model. Attempting to use this with a memory model other than huge will generate an error. Using `--BANK` is optional and overrides the defaults specified in the `8051-c.ini` file. It takes the form `--BANK=`*`base,size,start,num`* where:

- *`base`* is the logical starting address of the banked area. This and the next value, *`size`*, define a window which is mapped into ROM at various physical addresses.
- *`size`* is the size of the banked area which is mapped into ROM at various physical addresses.
- *`start`* is the bank number in which to commence placement of banked code. This parameter is an ordinary decimal number, and will in most cases be zero.
- *`num`* is the number of banks. This parameter is an ordinary decimal number.
- For example, to specify a bank window starting at 8000, ending at FFFF, with 16 banks:

```
--BANK=8000,8000,0,16
```

See Section 7.1.5 for information on how to specify identifier length when compiling within HI-TIDE.

10.4.20 --CHAR=*type*: Make Char Type Signed or Unsigned

Unless this option is used, the default behaviour of the compiler is to make all character values and variables of type `unsigned char` unless explicitly declared or cast to `signed char`. This option will make the default char type `signed char`. When using this option, any unsigned character object will have to be explicitly declared `unsigned char`.

The range of a `signed` character type is -128 to +127 and the range of similar `unsigned` objects is 0 to 255.

See Section 7.6.1 for information on how to specify the default `char` type when compiling within HI-TIDE.

10.4.21 --CHIP=*processor*: Define Processor

This option defines the processor which is being used. To see a list of supported processors that can be used with this option, use the `--CHIPINFO` option.

See Section 5.6.2 for information on how to specify a processor when compiling within HI-TIDE.

10.4.22 --CHIPINFO: Display a List of Supported Processors

The `--CHIPINFO` option simply displays a list of processors the compiler supports. The names listed are those chips defined in the chipinfo file and which may be used with the `--chip` option. All devices are listed in Section C in the Appendix.

10.4.23 --CODEOFFSET=*address*: Specify an Offset For Program Code

The `--CODEOFFSET` option is used to shift program entry from the default location by the specified address. Any code, or data associated with this code, that is explicitly linked at a particular address — this will be the reset vector and associate constants — will be shifted up by the address specified with this option.

Psects that are placed anywhere within a linker class are linked as normal. The `--ROM` option can be used in conjunction with the `--CODEOFFSET` to move all compiler output. These two options allow generation of code that is to be downloaded by a bootloader and needs to be executed from an address other than zero.

See Section 7.5.2 for information on how to specify an entry offset when compiling within HI-TIDE.

10.4.24 --CR=*file*: Generate Cross Reference Listing

The `--CR` option will produce a *cross reference listing*. If the *file* argument is omitted, the "raw" cross reference information will be left in a temporary file, leaving the user to run the `CREF` utility. If a filename is supplied, for example `--CR=test.crf`, `C51` will invoke `CREF` to process the cross reference information into the listing file, in this case `test.crf`. If multiple source files are to be included in the cross reference listing, all must be compiled and linked with the one `C51` command. For example, to generate a cross reference listing which includes the source modules `main.c`, `module1.c` and `nvram.c`, compile and link using the command:

```
C51 --CHIP=8051AH --CR=main.crf main.c module1.c nvram.c
```

10.4.25 --ERRFORMAT and --WARNFORMAT: Format For Compiler Messages

If the --ERRFORMAT option is not used, the default behaviour of the compiler is to display any errors in a "human readable" format line with a *caret* "^" and error message pointing out the offending characters in the source line, for example:

```
x.c: main()
     4: PORT_A = xFF;
                 ^ undefined identifier: xFF
```

This standard format is perfectly acceptable to a person reading the error output, but is not usable with environments which support compiler error handling. The following sections indicate how this option may be used in such situations.

10.4.25.1 Using the --ERRFORMAT and --WARNFORMAT Option

Using the these option instructs the compiler to generate error and warning messages in a format which is acceptable to some text editors and development environments.

If the same source code as used in the example above were compiled using the --ERRFORMAT option, the error output would be:

```
x.c 4 9: undefined identifier: xFF
```

indicating that the error occurred in file x.c at line 4, offset 9 characters into the statement. The second numeric value - the column number - is relative to the left-most non-space character on the source line. If an extra *space* or *tab* character were inserted at the start of the source line, the compiler would still report an error at line 4, column 9.

10.4.25.2 Modifying the Standard Format

If the default error and warning message format does not meet your editor's requirement, you can redefine its format by either using the ERRFORMAT and WARNFORMAT option or by setting two environment variables: HTC_ERR_FORMAT and HTC_WARN_FORMAT. These options are in the form of a printf-style string in which you can use the specifiers shown in Table 10.4. For example:

```
--ERRFORMAT="file %f; line %l; column %c; %s"
```

The column number is relative to the left-most non-space character on the source line.

The environment variables can be set in a similar way, for example setting the environment variables from within DOS can be done with the following DOS commands:

Table 10.4: Error format specifiers

Specifier	Expands To
%f	Filename
%l	Line number
%c	Column number
%s	Error string

```
set HTC_WARN_FORMAT=WARNING: file %f; line %l; column %c; %s
set HTC_ERR_FORMAT=ERROR: file %f; line %l; column %c; %s
```

Using the previous source code, the output from the compiler when using the above environment variables would be:

```
ERROR: file x.c; line 4; column 9; undefined identifier: xFF
```

Remember that if these environment variables are set in a batch file, you must prepend the specifiers with an additional *percent* character to stop the specifiers being interpreted immediately by DOS, e.g. the filename specifier would become `%f`.

10.4.26 --GETOPTION=*app,file*: Get Command Line Options

This option is used to retrieve the command line options which are used for named compiler application. The options are then saved into the given file. This option is not required for most projects.

10.4.27 --HELP<=*option*>: Display Help

The `--HELP` option displays information on the `C51` compiler options. To find out more about a particular option, use the option's name as a parameter. For example:

```
C51 --help=warn
```

This will display more detailed information about the `--WARN` option.

10.4.28 --IDE=*type*: Specify the IDE Being Used

This option is used to automatically configure the compiler for use by the named Integrated Development Environment (IDE). The supported IDE's are shown in Table 10.5.

Table 10.5: Supported IDEs

Suboption	**IDE**
`hitide`	HI-TECH Software's HI-TIDE

10.4.29 `--INTRAM=`*`address`*: Specify Internal RAM Address

This option defines the address in internal RAM where *auto* variables, function arguments, *idata* and *near* variables, collectively known as *internal* storage, will be located. The *intram* value should normally be set to address 20, starting user variables just above any *bit* variables. If this value is supplied as 0 or 20, the linker options will be configured to concatenate all internal storage onto the *bit* variables which always start at 20H. If a value of 21 or higher is used, *internal* storage will start at that address but *bit* variables will still start at 20H. Care should be taken to avoid overlaying internal storage over the *bit* variables. For example, there will be a clash if internal storage is linked at 21H and there are more than 8 *bit* variables.

See Section 7.3.3 for information on how to specify the internal RAM when compiling within HI-TIDE.

10.4.30 `--MEMMAP=`*`file`*: Display Memory Map

This option will display a memory map for the specified map file. This option is seldom required, but would be useful if the linker is being driven explicitly, i.e. instead of in the normal way through the driver. This command would display the memory summary which is normally produced at the end of compilation by the driver.

10.4.31 `--NOEXEC`: Do Not Execute Compiler

The `--NOEXEC` option causes the compiler to go through all the compilation steps, but without actually performing any compilation or producing any output. This is often useful when used in conjunction with the `-V` (verbose) option in order to see all of the command lines the compiler uses to drive the compiler applications.

10.4.32 `--NOPS`: Insert Debug NOPs

This option is intended for debuggers which use the `LCALL` instruction to implement a breakpoint. It will cause `NOP` instructions to be strategically inserted into the object code, the purpose of which is to reserve space for an `LCALL` to be inserted at a label. Because this causes differences in the object

code generated, this option is separate from, and should be supplied in addition to, the -G option, if desired.

See Section 7.1.6 for information on how to specify an entry offset when compiling within HI-TIDE.

10.4.33 --NVRAM=*address*: Specify Non-volatile RAM Address

This option defines the address in external RAM of a non-volatile RAM area used to store *persistent* variables. If this feature is not used or if all RAM is non-volatile then this option should not be specified. See Section 11.3.9.1 for more information on *persistent* variables, and Section 7.3.4 for information on specifying the non-volatile RAM location when compiling from HI-TIDE.

10.4.34 --OPT<=*type*>: Invoke Compiler Optimizations

The --OPT option allows control of all the compiler optimizers. By default, without this option, all optimizations are enabled. The options --OPT or --OPT=all also enable all optimizations. Optimizations may be disabled by using --OPT=none, or individual optimizers may be controlled, e.g. --OPT=as_all will only enable the assembler optimizer. The options --OPT=speed or --OPT=space can also be used to control optimizations for speed or space accordingly.

See Section 7.1.2 and 7.1.3 for information on how to use the optimizers when compiling within HI-TIDE.

10.4.35 --OUTDIR=*directory*: Specify Output Directory

This option allows control over the directory which output files from the compiler are placed. If no --OUTDIR option is specified, the current working directory is used. Note that the directory specified by the --OUTDIR option may be superseded by that specified by the -O option for any files produced by the linker or subsequently run applications.

10.4.36 --OUTPUT=*type*: Specify Output File Type

This option allows the type of the output file to be specified. If no --OUTPUT option is specified, the output file's name will be derived from the first source or object file specified on the command line. The available output file format are shown in Table 10.6.

See Section 7.4.1 for information on how to specify the output file type when compiling within HI-TIDE.

Table 10.6: Output file formats

option name	File format
intel	*Intel* HEX
tek	Tektronic
aahex	*American Automation* symbolic HEX file
mot	*Motorola* S19 HEX file
ubrof	UBROF format
bin	Binary file
cof	Common Object File Format
elf	ELF/DWARF file format
omf51	OMF-51 format
eomf51	Extended OMF-51 format

10.4.37 --PRE: Produce Preprocessed Source Code

The `--PRE` option is used to generate preprocessed C source files with an extension `.pre`. This may be useful to ensure that preprocessor macros have expanded to what you think they should. Use of this option can also create C source files which do not require any separate header files. This is useful when sending files for technical support.

See Section 4.1.1.3 for information on how to produce preprocessed files when compiling within HI-TIDE.

10.4.38 --PROTO: Generate Prototypes

The `--PROTO` option is used to generate `.pro` files containing both ANSI and K&R style function declarations for all functions within the specified source files. Each `.pro` file produced will have the same base name as the corresponding source file. Prototype files contain both ANSI C-style prototypes and old-style C function declarations within conditional compilation blocks.

The extern declarations from each `.pro` file should be edited into a global header file which is included in all the source files comprising a project. The `.pro` files may also contain static declarations for functions which are local to a source file. These static declarations should be edited into the start of the source file. To demonstrate the operation of the `--PROTO` option, enter the following source code as file `test.c`:

```
#include <stdio.h>
add(arg1, arg2)
```

```
int *   arg1;
int *   arg2;
{
        return *arg1 + *arg2;
}

void printlist(int * list, int count)
{
        while (count--)
                printf("%d ", *list++);
        putchar('\n');
}
```

If compiled with the command:

```
C51 --CHIP=8051AH --PROTO test.c
```

C51 will produce `test.pro` containing the following declarations which may then be edited as necessary:

```
/* Prototypes from test.c */
/* extern functions - include these in a header file */
#if     PROTOTYPES
extern int add(int *, int *);
extern void printlist(int *, int);
#else/* PROTOTYPES */
extern int add();
extern void printlist();
#endif/* PROTOTYPES */
```

10.4.39 --RAM=*lo-hi,<lo-hi,...>*: Specify Additional RAM Ranges

This option is used to specify memory, in addition to any RAM specified in the chipinfo file, which should be treated as available RAM space. Strictly speaking, this option specifies the areas of memory that may be used by writable (RAM-based) objects, and not necessarily those areas of memory which contain physical RAM. The output that will be placed in the ranges specified by this option are typically variables that a program defines.

Some chips have an area of RAM that can be remapped in terms of its location in the memory space. This, along with any fixed RAM memory defined in the chipinfo file, are grouped an made available for RAM-based objects.

For example, to specify an additional range of memory to that present on-chip, use:

```
--RAM=default,+1000-2fff
```

for example. To only use an external range and ignore any on-chip memory, use:

```
--RAM=1000-2fff
```

This option may also be used to reserve memory ranges already defined as on-chip memory in the chipinfo file. To do this supply a range prefixed with a *minus* character, `-`, for example:

```
--RAM=default,-100-103
```

will use all the defined on-chip memory, but not use the addresses in the range from 100h to 103h for allocation of RAM objects.

See Section 7.3.2 for information on how to specify data memory ranges when compiling within HI-TIDE.

10.4.40 **--ROM=*lo-hi,<lo-hi,...>|tag*:** Specify Additional ROM Ranges

This option is used to specify memory, in addition to any ROM specified in the chipinfo file, which should be treated as available ROM space. Strictly speaking, this option specifies the areas of memory that may be used by read-only (ROM-based) objects, and not necessarily those areas of memory which contain physical ROM. The output that will be placed in the ranges specified by this option are typically executable code and any data variables that are qualified as `const`.

When producing code that may be downloaded into a system via a bootloader the destination memory may indeed by some sort of (volatile) RAM. To only use on-chip ROM memory, this option is not required. For example, to specify an additional range of memory to that on-chip, use:

```
--ROM=default,+1000-2fff
```

for example. To only use an external range and ignore any on-chip memory, use:

```
--ROM=1000-2fff
```

For those chips that have an area of RAM that can be remapped in terms of its location in the memory space, the tag `remapram` may be used to indicate that the destination is the remapable area, for example:

```
--ROM=remapram
```

This option may also be used to reserve memory ranges already defined as on-chip memory in the chipinfo file. To do this supply a range prefixed with a *minus* character, `-`, for example:

Table 10.7: Runtime environment suboptions

Suboption	Controls	On (+) implies
`init`	The code present in the startup module that copies the `data` psect's ROM-image to RAM.	The `data` psect's ROM image is copied into RAM.
`clib`	The inclusion of library files into the output code by the linker.	Library files are linked into the output.
`clear`	The code present in the startup module that clears the `bss` psects.	The `bss` psect is cleared.
`stack`	The code present in the startup module that initializes the stack pointer.	The stack pointer is initialized.
`keep`	Whether the startup module source file is deleted after compilation.	The startup module is not deleted.
`no_startup`	Whether a startup module is produced and linked into the output.	The startup module is not generated or linked into the output.

```
--ROM=default,-1000-1fff
```

will use all the defined on-chip memory, but not use the addresses in the range from 1000h to 1fffh for allocation of ROM objects.

See Section 7.3.1 for information on how to specify program memory ranges when compiling within HI-TIDE.

10.4.41 `--RUNTIME=`*`type`*: Specify Runtime Environment

The `--RUNTIME` option is used to control what is included as part of the runtime environment. The runtime environment encapsulates any code that is present at runtime which has not been defined by the user, instead supplied by the compiler, typically as library code.

All runtime features are enabled by default and this option is not required for normal compilation. The usable suboptions include those shown in Table 10.7.

See Section 7.5.1 for information on how to specify runtime environment options when compiling within HI-TIDE.

10.4.42 **--SCANDEP**: Scan For Dependencies

When this option is used, a `.dep` (dependency) file is generated. The dependency file lists those files on which the source file is dependant. Dependencies result when one file is `#include`d into another.

10.4.43 **--SETOPTION=*app,file***: Set the Command Line Options For Application

This option is used to supply alternative command line options for the named application when compiling. This option is not required for most projects.

10.4.44 **--STRICT**: Strict ANSI Conformance

The `--STRICT` option is used to enable strict ANSI conformance of all special keywords. HI-TECH C supports various special keywords (for example the `persistent` type qualifier). If the `--STRICT` option is used, these keywords are changed to include two *underscore* characters at the beginning of the keyword (e.g. `__persistent`) so as to strictly conform to the ANSI standard. Be warned that use of this option may cause problems with some standard header files (e.g. `<intrpt.h>`).

See Section 7.6.3 for information on ANSI conformance when compiling within HI-TIDE.

10.4.45 **--SUMMARY=*type***: Select Memory Summary Output Type

Use this option to select the type of memory summary that is displayed after compilation. By default, or if the `mem` suboption is selected, a memory summary is shown. This shows the memory usage for all available linker classes.

A psect summary may be shown by enabling the `psect` suboption. This shows individual psects, after they have been grouped by the linker, and the memory ranges they cover.

See Section 8.3.4 for information on compile summaries produced when compiling within HI-TIDE.

10.4.46 **--VER**: Display the Compiler's Version Information

The `--VER` option will display what version of the compiler is running.

10.4.47 --WARN=*level*: Set Warning Level

The --WARN option is used to set the compiler warning level. Allowable warning levels range from -9 to 9. The warning level determines how pedantic the compiler is about dubious type conversions and constructs. The default warning level --WARN=lvl0 will allow all normal warning messages. Warning level --WARN=lvl1 will suppress the message Func() declared implicit int. --WARN=lvl3 is recommended for compiling code originally written with other, less strict, compilers. --WARN=lvl9 will suppress all warning messages. Negative warning levels --WARN=lvl-1, --WARN=lvl-2 and --WARN=lvl-3 enable special warning messages including compile-time checking of arguments to printf() against the format string specified.

Use this option with care as some warning messages indicate code that is likely to fail during execution, or compromise portability.

See Section 7.1.1.1 for information on how to specify the warning level when compiling within HI-TIDE.

Chapter 11

C Language Features

HI-TECH C supports a number of special features and extensions to the C language which are designed ease the task of producing ROM based applications. This chapter documents the compiler options and special features which are available.

11.1 Files

11.1.1 Source Files

The extension used with source files is important as it is used by the compiler drivers to determine their content. Source files containing C code should have the extension `.c`, assembler files should have extensions of `.as`, relocatable object files require the `.obj` extension, and library files should be named with a `.lib` extension.

11.1.2 Symbol files

The C51 -G and -H options tell the compiler to produce a symbol file which can be used by debuggers and simulators to perform symbolic and source level debugging. The -H option produces symbol files which contain only assembler level information, whereas the -G option also includes C source level information. If no symbol file name is specified, by default a file called *l.sym* will be produced. For example, to produce a symbol file called test.sym which includes C source level information use:

```
c51 -8051 -Gtest.sym test.c
```

The UBROF output file format which can be produced by the compiler contains both object code and symbolic debug information and should be used in preference to separate symbol files if you have an in-circuit emulator which supports it.

11.1.3 Standard Libraries

C51 includes a number of standard libraries, each with the range of functions described in Chapter A. The naming convention used for the standard libraries is in the form `51`***pbml***`.lib`. The meaning of each field is described here, where:

p Represents the processor Architecture which is `-` for the generic 8051, `7` for the 80C517 and derivatives, and `a` for the 80C751 based processors which lack the `LJMP` and `LCALL` instructions (small model available only).

b Represents the banking Scheme and is a letter representing the type of banking (`bcall.as` module, see Section 11.5.3) used in huge model. For all other memory models, and for the generic 8051 huge model library, this will be the reserved letter `N`. This letter corresponds to the BANKTYPE entry in the chipinfo file. For more information, see the following Section 11.2.1.

m Represents the memory model is one of `s`, m, `l`, or `h`, which represent the small, medium, large, and huge models, respectively.

l Represents the library yype and is `c` for standard library, `l` for the library which contains only printf-related functions with additional support for longs, and `f` for the library which contains only printf-related functions with additional support for longs and floats.

11.1.4 Run-time Startup Module

A C program requires certain objects to be initialised and the processor to be in a particular state before it can begin execution of its function `main()`. It is the job of the *runtime startup* code to perform these tasks.

Traditionally, runtime startup code is a generic, precompiled routine which is always linked into a user's program. Even if a user's program does not need all aspects of the runtime startup code, redundant code is linked in which, albeit not harmful, takes up memory and slows execution. For example, if a program does not use any uninitialized variables, then no routine is required to clear the `bss` psects.

HI-TECH C differs from other compilers by using a novel method to determine exactly what runtime startup code is required and links this into the program automatically. It does this by performing an additional link step which does not produce any usable output, but which can be used to determine the requirements of the program. From this information HI-TECH C then "writes" the

assembler code which will perform the runtime startup. This code is stored into a file which can then be assembled and linked into the remainder of the program in the usual way.

Since the runtime startup code is generated automatically on every compilation, the generated files associated with this process are deleted after they have been used. If required, the assembler file which contains the runtime startup code can be kept after compilation and linking by using the driver option --RUNTIME=keep. The residual file will be called `startup.as` and will be located in the current working directory. If you are using an IDE to perform the compilation the destination directory is dictated by the IDE itself, however you may use the `--OUTDIR` option to specify an explicit output directory to the compiler. If the runtime startup module is not deleted and source-level debug information is enabled, then source-level debug information is produced for the runtime startup module. This allows the user to step through this module within an IDE.

This is an automatic process which does not require any user interaction, however some aspects of the runtime code can be controlled, if required, using the `--RUNTIME` option. These are described in the sections below.

11.1.4.1 Stack Initialization

The `stack` suboption to the `--RUNTIME` option allows control over the initialization of the stack pointer. By default, the stack pointer is initialized by the runtime startup code. The stack pointer is set to an address equal to the lower bound of a psect called `stack`. This psect is normally empty, but is used as a placeholder to mark the starting position of the stack pointer.

To disable initialization of the stack pointer, disable the `stack` suboption, e.g. `--RUNTIME=default,-stack`, which specifies the default runtime startup code functionality, excluding the stack initialization.

Changing the address at which the stack pointer is initialized is *not* handled by this option, but can be altered by linking the `stack` psect at the required location in memory. This allows much better control over placement of the stack with respect to other RAM-based objects, which may appear at different locations as the program changes. By default, the `stack` psect is placed at the top of RAM, which allows maximum growth of the stack downwards in memory. Adjustment of the linker options can be made by using the `-L` command-line driver option, see 10.4.8 or using the HI-TIDE memory options described in the 7.3.2.

11.1.4.2 Initialization of Data Psects

Another job of the runtime startup code is ensure that any initialized variables contain their initial value before the program begins execution. Initialized variables are those which are not `auto` objects and which are assigned an initial value in their definition, for example `input` in the following example.

```
int input = 88;
void main(void) { ...
```

Such initialized objects are placed within the `data` psect. These psects have two components. The first is an area which contains the initial values. This is positioned in non-volatile memory at an address known as the *load address*. The other component is where the variables will reside, and be accessed, once the program is executing. This area is positioned in RAM at an address known as the *link address*. The runtime startup code performs a block copy of the values from the load address to the link address.

The block copy of the `data` psect may be omitted by disabling the `init` suboption of `--RUNTIME`. For example:

```
--RUNTIME=default,-init
```

With this part of the runtime startup code absent, the contents of initialized variables will be unpredictable when the program begins execution.

Variables whose contents should be preserved over a reset, or even power off, should be qualified with persistent. Such variables are linked at a different area of memory and are not altered by the runtime startup code in any way.

11.1.4.3 Clearing the Bss Psects

The ANSI standard dictates that those non-`auto` objects which are not initialized must be cleared before execution of the program begins. The compiler does this by grouping all such uninitialized objects into the `bss` psect. This psect is then cleared as a block by the runtime startup code.

The block clear of the `bss` psect may be omitted by disabling the `clear` suboption of `--RUNTIME`. For example:

```
--RUNTIME=default,-clear
```

With this part of the runtime startup code absent, the contents of uninitialized variables will be unpredictable when the program begins execution.

Variables whose contents should be preserved over a reset, or even power off, should be qualified with persistent. Such variables are linked at a different area of memory and are not altered by the runtime startup code in anyway.

11.1.4.4 Linking in the C Libraries

By default, a set of libraries are automatically passed to the linker to be linked in with user's program. The libraries can be omitted by disabling the `clib` suboption of `--RUNTIME`. For example:

```
--RUNTIME=default,-clib
```

With this part of the runtime startup code absent, the user must provide alternative library or source files to allow calls to library routines. This suboption may be useful if alternative library or source files are available and you wish to ensure that no HI-TECH C library routines are present in the final output.

11.1.4.5 Executing the Main Function

The last code executed as part of the runtime startup code is that to call the `main()` function — the first user-defined C function in a program.

11.1.5 The *powerup* Routine

Some hardware configurations require special initialization, often within the first few cycles of execution after reset. To achieve this there is a hook to the reset vector provided via the *powerup* routine. This is a user-supplied assembler module that will be executed immediately on reset. A "dummy" powerup routine is included in the file `powerup.as`. This file can be copied, modified and included into your project to replace the default (empty) powerup routine that is present in the standard libraries.

If you use a powerup routine, you will need to add a jump to the `start1` label after your initializations. Refer to comments in the powerup source file for further details.

If the powerup routine is included into a project — specifically if the `powerup` psect is of non-zero length — the runtime startup module will define a reset vector that points to this powerup routine.

11.2 Processor-related Features

11.2.1 Processor Support

C51 currently supports many hundreds of 8051 derivatives. Additional code-compatible processors may be added by editing the `8051-c.ini` file in the LIB directory. User-defined processors should be placed at the end of the file. The header of the file explains how to specify a processor. Newly added processors will be available the next time you compile by selecting the name of the new processor on the command line in the usual way.

11.3 Supported Data Types

The 8051 compiler supports basic data types of 1, 2 and 4 byte size. All multi-byte types follow *most* significant byte first format, also known as *big endian*. Word size values thus have the most

Table 11.1: Basic data types

Type	Size (in bits)	Arithmetic Type
`bit`	1	boolean
`char`	8	signed or unsigned integer[1]
`unsigned char`	8	unsigned integer
`short`	16	signed integer
`unsigned short`	16	unsigned integer
`int`	16	signed integer
`unsigned int`	16	unsigned integer
`long`	32	signed integer
`unsigned long`	32	unsigned integer
`float`	32	real
`double`	32	real

significant byte at the lower address, and double word size values have the most significant byte and most significant word at the lowest address. The 8051 is a byte oriented machine, there are no alignment restrictions on word or long sized objects. Structures and structure elements are also free of alignment restrictions, thus structures will never contain "holes".

Note that when right-shifting an integer data type, a zero is places in the most significant bit, whether the integer is signed or unsigned.

Table 11.1 shows the data types and their corresponding size and arithmetic type.

11.3.1 Radix Specifiers and Constants

The format of integral constants specifies their radix. `C51` supports the ANSI standard radix specifiers as well as one which enables binary constants to specified in C code. The format used to specify the radices are given in Table 11.2. The letters used to specify binary or hexadecimal radices are case insensitive, as are the letters used to specify the hexadecimal digits.

Any integral constant will have a type which is the smallest type that can hold the value without overflow. The suffix `l` or `L` may be used with the constant to indicate that it must be assigned either a `signed long` or `unsigned long` type, and the suffix `u` or `U` may be used with the constant to indicate that it must be assigned an unsigned type, and both `l` or `L` and `u` or `U` may be used to indicate `unsigned` long int type.

Floating-point constants have `double` type unless suffixed by `f` or `F`, in which case it is a `float` constant. The suffixes `l` or `L` specify a `long double` type which is considered an identical type to `double` by `C51`.

Table 11.2: Radix formats

Radix	Format	Example
binary	0b*number* or 0B*number*	0b10011010
octal	0*number*	0763
decimal	number	129
hexadecimal	0x*number* or 0X*number*	0x2F

Character constants are enclosed by single quote characters “ ' ”, for example `'a'`. A character constant has char type. Multi-byte character constants are not supported.

String constants or string literals are enclosed by double quote characters “ " “, for example `"hello` world”. The type of string constants is `const char *` and the strings are stored in ROM. Assigning a string constant to a non-`const` char pointer will generate a warning from the compiler. For example:

```
char * cp          = "one";     // "one" in ROM, produces warning
const char * ccp   = "two";     // "two" in ROM
char ca[]          = "two";     // "two" different to the above
```

A non-`const` array initialised with a string, for example the last statement in the above example, produces an array in RAM which is initialised at startup time with the string `"two"` (copied from ROM), whereas a constant string used in other contexts represents an unnamed `const`-qualified array, accessed directly in ROM.

C51 will use the same storage location and label for strings that have identical character sequences, except where the strings are used to initialise an array residing in RAM as indicated in the last statement in the above example.

Two adjacent string constants (i.e. two strings separated *only* by white space) are concatenated by the compiler. Thus:

```
const char * cp = "hello " "world";
```

assigned the pointer with the string ”`hello world"`.

11.3.2 Bit Data Types

HI-TECH C allows single bit variables to be declared using the keyword *bit*. A variable declared *bit*, for example:

```
static bit init_flag;
```

will be allocated in the bit addressable psect *rbit*, and will be visible only in that module or function. When the following declaration is used outside any function:

```
bit init_flag;
```

init_flag will be globally visible.

The *rbit* psect is linked into the 8051 bit addressable area from 20H to 2FH, limiting the number of *bit* variables in a single program to 128.

Bit variables are manipulated using the efficient 8051 bit addressing modes. These variables behave in all respects like normal unsigned char variables, except that they may only contain the values 0 and 1, therefore they provide a convenient and efficient method of storing boolean flags without consuming large amounts of internal RAM. Due to the lack of suitable addressing modes on the 8051 it is not possible to declare pointers to *bit* variables or to statically initialise *bit* variables. If the C51 flag -STRICT is used, the *bit* keyword becomes *__bit*.

11.3.2.1 Using Bit-Addressable Registers

The *bit* variable facility may be combined with absolute variable declarations to access the bit addressable special function registers at bit addresses 80H to FFH. The 128 bit addresses from 80H to FFH map onto the 16 special function registers with addresses divisible by 8. Thus individual bits in function registers at addresses 80H, 88H ... F8H may be accessed. For each bit addressable SFR, address of bit 0 of the special function register is the same as its byte address. Thus, bit 0 of the SFR at address A8H is bit address A8H, bit 1 is A9H, up to bit 7 which is AFH. For example, to access bit 3 of port P2 at A0H, declare a *bit* variable at absolute address A3H:

```
static bit     P2_3 @ 0xA3;
```

Similarly, bits 0 to 7 of port P0 at address 80H would be declared as:

```
static bit     P0_0 @ 0x80;
static bit     P0_1 @ 0x81;
static bit     P0_2 @ 0x82;
static bit     P0_3 @ 0x83;
static bit     P0_4 @ 0x84;
static bit     P0_5 @ 0x85;
static bit     P0_6 @ 0x86;
static bit     P0_7 @ 0x87;
```

11.3.3 8-Bit Data Types

HI-TECH C supports both *signed char* and *unsigned char* 8 bit integral types. The default *char* type is *signed char* unless the C51 option –char=unsigned is used, in which case it is *unsigned char*. *Signed char* is an 8 bit two's complement signed integer type, representing integral values from -128 to +127 inclusive. *Unsigned char* is an 8 bit unsigned integer type, representing integral values from 0 to 255 inclusive.

It is a common misconception that the C *char* types are intended purely for ASCII character manipulation. This is not true, indeed the C language makes no guarantee that the default character representation is even ASCII. The *char* types are simply the smallest of up to four possible integer sizes, and behave in all respects like integers. The reason for the name *char* is historical and does not mean that *char* can only be used to represent characters. It is possible to freely mix *char* values with *short*, *int* and *long* in C expressions.

On the 8051 the *char* types will commonly be used for a number of purposes, as 8 bit integers, as storage for ASCII characters, and for access to I/O locations. *Unsigned char* is the C type which logically maps onto the format of most 8051 special function registers. The *unsigned char* type is the most efficient data type on the 8051 and maps directly onto the 8 bit bytes which are most efficiently manipulated by 8051 instructions. It is suggested that *char* types be used wherever possible so as to maximize performance and minimize code size.

11.3.4 16-Bit Data Types

HI-TECH C supports four 16 bit integer types. *Int* and *short* are 16 bit two's complement signed integer types, representing integral values from -32768 to +32767 inclusive; *Unsigned int* and *unsigned short* are 16 bit unsigned integer types, representing integral values from 0 to 65535 inclusive. 16 bit integer values are represented in *big endian* format with the most significant byte at the lower address.

Both *int* and *short* are 16 bits wide as this is the smallest integer size allowed by the ANSI standard for C. 16 bit integers were chosen so as not to violate the ANSI standard. Allowing a smaller integer size, such as 8 bits would lead to a serious incompatibility with the C standard. 8 bit integers are already fully supported by the *char* types and should be used in place of *int* wherever possible.

11.3.5 32-Bit Data Types

HI-TECH C supports two 32 bit integer types. *Long* is a 32 bit two's complement signed integer type, representing integral values from -2147483648 to +2147483647 inclusive. *Unsigned long* is a 32 bit unsigned integer type, representing integral values from 0 to 4294967295 inclusive. 32 bit integer values are represented in *big endian* format with the most significant word and most significant byte

Table 11.3: Floating-point formats

Format	Sign	biased exponent	mantissa
IEEE 754 32-bit	x	xxxx xxxx	xxx xxxx xxxx xxxx xxxx xxxx

Table 11.4: Floating-point format example IEEE 754

Number	biased exponent	1.mantissa	decimal
7DA6B69Bh	11111011b	1.01001101011011010011011b	2.77000e+37
	(251)	(1.302447676659)	

at the lowest address. 32 bits are used for *long* and *unsigned long* as this is the smallest long integer size allowed by the ANSI standard for C. It is suggested that 32 bit integers be used sparingly due to the code size and speed penalty imposed by 32 bit integer manipulation on a simple 8 bit architecture like the 8051.

11.3.6 Floating Point Types and Variables

Floating point is implemented using the IEEE 754 32-bit format.

The 32-bit format is used for all `float` and `double` values.

This format is described in Table 11.3, where:

- sign is the sign bit
- The exponent is 8-bits which is stored as *excess 127* (i.e. an exponent of 0 is stored as 127).
- mantissa is the mantissa, which is to the right of the radix point. There is an implied bit to the left of the radix point which is always 1 except for a zero value, where the implied bit is zero. A zero value is indicated by a zero exponent.

The value of this number is $(-1)^{sign}$ x $2^{(exponent-127)}$ x 1.mantissa.

Here are some examples of the IEEE 754 32-bit formats:

Note that the most significant bit of the mantissa column in Table 11.4 (that is the bit to the left of the radix point) is the implied bit, which is assumed to be 1 unless the exponent is zero (in which case the float is zero).

The 32-bit example in Table 11.4 can be calculated manually as follows.

The sign bit is zero; the biased exponent is 251, so the exponent is 251-127=124. Take the binary number to the right of the decimal point in the mantissa. Convert this to decimal and divide it by 2^{23} where 23 is the number of bits taken up by the mantissa, to give 0.302447676659. Add one to this fraction. The floating-point number is then given by:

$-1^0 \times 2^{124} \times 1.302447676659 = 1 \times 2.126764793256e+37 \times 1.302447676659 \approx 2.77000e+37$

Variables may be declared using the `float` and `double` keywords, respectively, to hold values of these types. Floating point types are always signed and the `unsigned` keyword is illegal when specifying a floating point type. Types declared as `long double` will use the same format as types declared as `double`.

11.3.7 Structures and Unions

HI-TECH C supports *struct* and *union* types of any size from one byte upwards. Structures and unions may be passed freely as function arguments and return values. Pointers to structures and unions are fully supported. The 8051 is a byte oriented machine, so there are no alignment restrictions on structure and union members.

11.3.7.1 Bit Fields in Structures

HI-TECH C fully supports *bit fields* in structures. Version 7 and later of the compiler allocate bit fields starting with the most significant bit. Bit fields are allocated within 16 bit words, the first bit allocated will be the most significant bit of the most significant byte of the word, corresponding to the sign bit in a signed 16 bit word. Bit fields are always allocated in 16 bit units, starting from the most significant bit. When a bit field is declared, it is allocated within the current 16 bit unit if it will fit. Otherwise a new 16 bit word is allocated within the structure. Bit fields never cross the boundary between 16 bit words, but may span the byte boundary within a given 16 bit allocation unit.

For example, the declaration:

```
struct {
        unsigned        hi : 1;
        unsigned        dummy : 14;
        unsigned        lo : 1;
} foo @ 0x10;
```

will produce a structure occupying 2 bytes from address 10h. The field *hi* will be bit 15 of address 10h, lo will be bit 0 of address 11h. The most significant bit of *dummy* will be bit 14 of address 10h and the least significant bit of *dummy* will be bit 1 of address 11h. Unnamed bit fields may be declared to pad out unused space between active bits in control registers. For example, if *dummy* is never used the structure above could have been declared as:

```
struct {
        unsigned        hi : 1;
        unsigned           : 14;
        unsigned        lo : 1;
} foo @ 0x10;
```

11.3.8 Standard Type Qualifiers

11.3.8.1 Const and Volatile Type Qualifiers

HI-TECH C supports the use of the ANSI type qualifier *const* and *volatile*. The *const* type qualifier is used to tell the compiler that an object has a constant value and will not be modified. If any attempt is made to modify an object declared *const*, the compiler will issue a warning. User defined objects declared const are placed in a special *psect* called *const*. For example:

```
const int       version = 3;
```

The *volatile* type qualifier is used to tell the compiler that an object cannot be guaranteed to retain its value between successive accesses. This prevents the optimizer from eliminating apparently redundant references to objects declared *volatile* because it may alter the behaviour of the program to do so. All I/O ports and any variables which may be modified by interrupt routines should be declared *volatile*, for example:

```
volatile unsigned char P1 @ 0x90;
```

11.3.9 Special Type Qualifiers

HI-TECH C supports special type qualifiers, *persistent*, *near*, *far*, *code* and *idata* to allow the user to control placement of *static* and *extern* class variables into particular address spaces.

If the C51 –STRICT option is used, these type qualifiers are changed to __*persistent*, __near, __*far*, __*code* and __*idata*. These type qualifiers may also be applied to pointers, *near* and *idata* allow the declaration of 8 bit pointers which use the register indirect addressing mode to access internal RAM.

These type qualifiers may not be used on variables of class *auto*. If used on variables local to a function they must be combined with the *static* keyword. You may not write:

```
void func(void)
{
near int intvar;     /* WRONG! */
.. other code ..
}
```

This is because *intvar* is of class *auto*. To declare *intvar* as a *near* variable local to function *test()*, write:

```
static near int intvar;.
```

11.3.9.1 Persistent Type Qualifier

By default, any C variables that are not explicitly initialized are cleared to zero on startup. This is consistent with the definition of the C language. However, there are occasions where it is desired for some data to be preserved across resets or even power cycles (on-off-on). The *persistent* type qualifier is used to qualify variables that should not be cleared on startup. In addition, any *persistent* variables will be stored in a different area of memory to other variables, and this area of memory may be assigned to a specific address (with the `--NVRAM` option to C51). Thus if a small amount of non-volatile RAM is provided then *persistent* variables may be assigned to that memory. On the other hand if all memory is non-volatile, you may choose to have persistent variables allocated to addresses by the compiler along with other variables (but they will still not be cleared). One advantage of assigning an explicit address for persistent variables is that this can remain fixed even if you change the program, and other variables get allocated to different addresses. This would allow configuration information etc. to be preserved across a firmware upgrade. Note that persistent variables are always allocated in external data memory, so in small model they will be treated as *far*.

There are some library routines provided to check and initialize persistent data - see the functions persist_check() and persist_validate() in the library functions cahpter A for more information.

11.3.9.2 Near Type Qualifier

The *near* type qualifier is used to place variables in internal RAM, where they may be more efficiently manipulated using 8051 instructions. This type qualifier is of most use in the medium, large, and huge models which place static variables in external RAM by default. In the small model all *static* and *extern* variables are placed in internal RAM so *near* need not be used. Variables declared to be *near* are placed in the psect *rbss*, which is linked into internal RAM in all memory models.

Use of *near* can provide substantial improvements to code quality, as access to external RAM is very inefficient due to the nature of the 8051 instruction set. If *intvar* is an *int* in external RAM, the statement intvar = 10; will generate code:

```
MOV     DPTR,#_intvar
MOV     A,#0
MOVX    @DPTR,A
INC     DPTR
MOV     A,#10
MOVX    @DPTR,a
```

If *intvar* is declared as *near int*, the same statement will generate:

```
MOV     _intvar,#high(10)
MOV     _intvar+1,#low(10)
```

Near variables may be statically initialized, for example:

```
static near int   bufsize = 128;
```

Initialized *near* variables such as *bufsize* will be placed in a psect called *rdata* and copied from ROM to internal RAM by the run-time startup module. *Near* can also be applied to pointers. A pointer of class *near*, for example:

```
near char *     nptr;
```

is a single byte pointer which can only address objects in the range 00H to FFH.

Near pointers are much more efficient than the 16 bit pointer classes and should be used wherever possible to maximize code efficiency. The compiler treats accesses via constant *near* pointers in a special manner, generating instructions which directly address the internal addresses. For example, the statement:

```
var = *(near char *)0x90;
```

will generate this code:

```
MOV     _var,90H.
```

This behaviour means that de-referencing a constant *near* pointer in the range 80H to FFH will access the SFR address space, not the indirect RAM area. If a *near* pointer variable containing 90H is de-referenced, the register indirect addressing mode will be used and internal RAM location 90H will be accessed, not SFR 90H.

11.3.9.3 Idata Type Qualifier

Idata is similar to *near*, except it declares variables which will always be accessed using the register indirect addressing mode of the 8051. *Near* variables must always reside at addresses below 80H, as direct addressing above 7FH accesses special function registers. *Idata* variables, which do not suffer from this limitation, can be used to access the indirectly addressable internal RAM area from 80H to FFH. For example, the declaration:

```
idata int       intvar;
```

creates a variable called *intvar* which is always accessed indirectly.

The statement:

```
intvar = 0x1234;
```

will generate this code:

```
MOV     R0,#_intvar
MOV     @R0,#12H
INC     R0
MOV     @R0,#34H
```

Statically initialized *idata* variables may be declared, such as:

```
idata int    count = 10;
```

Initialized *idata* variables are allocated in a psect called *irdata* which is copied from ROM to internal RAM by the startup code. Pointers to *idata* may be declared, for example:

```
idata char *    iptr;
```

Pointers to *idata* such as *iptr* occupy only a single byte of storage and are capable of addressing any object in internal RAM from 00H to FFH using the register indirect addressing mode.

The statement `ch = *iptr;` would generate this code:

```
MOV     R0,_iptr
MOV     _ch,@R0
```

If a constant *idata* pointer is de-referenced, the compiler will load the constant address into R0 or R1 and use the indirect addressing mode. Unlike a constant *near* pointer, *idata* pointers can never be used to access special function registers. For example, the statement:

```
var = *(idata char *)0x90;
```

will generate this code:

```
MOV      R0,#90H
MOV      _var,@R0
```

11.3.9.4 Far Type Qualifier

The type qualifier *far* is used to place objects in external RAM (the XDATA address space). This type qualifier is of the most use in the *small* memory model where all variables are placed in internal RAM unless declared *far*. The *medium*, *large*, and *huge* models already use external RAM for all *static* and *extern* variables, if using these models you will not need to use *far*. The *far* type qualifier is used to declare static variables as follows:

```
far int  f_int;
```

All accesses to *f_int* will use the MOVX instruction to access the XDATA address space. For example, the statement:

```
intvar = f_int;
```

will generate the code:

```
MOV     DPTR,#_f_int
MOVX    A,@DPTR
MOV     _intvar,A
INC     DPTR
MOVX    A,@DPTR
MOV     _intvar+1,A
```

Far may also be used to declare variables at absolute locations in the external address space, for example:

```
far unsigned char  sio_a_cmd @ 0xFF00;
far unsigned char  sio_a_data @ 0xFF01;
```

This will declare two externally mapped I/O ports at external addresses FF00H and FF01H. Pointers to objects of class *far* may be declared:

```
far char *      fptr;
```

Far pointers can access a "combined" address space which is the concatenation of internal RAM from 00H to FFH and external RAM from 100H to FFFFH. Accesses via *far* pointers generate code (or library calls) which check the high byte of the pointer and access internal RAM or external RAM as appropriate. For example, the statement `*fptr = 0;` will write 0 to internal ram address 6EH if *fptr* contains 006EH. If *fptr* contains 016EH, the same statement will write 0 to external RAM address 016EH.

11.3.9.5 Code Type Qualifier

The *code* type qualifier is used to place initialized static objects into the CODE address space of the 8051. Objects declared to be *code* must be statically initialized, for example:

```
code int  count = 0x1234;
```

The compiler will place *count* in the *code* psect, which is linked into program ROM immediately after the *text* psect. *Code* objects can not be modified, indeed the 8051 instruction set does not even allow the CODE address space to be written. The statement `icount = count;` will generate this code:

```
MOV     DPTR,#_count
CLR     A
MOVC    A,@A+DPTR
MOV     _icount,A
INC     DPTR
CLR     A
MOVC    A,@A+DPTR
MOV     _icount+1,A
```

All access to *code* objects takes place via the `MOVC A,@A+DPTR` instruction. Objects of class *code* occupy a completely separate address space to normal variables and constants. Standard pointers and *far* pointers cannot even address objects of class *code* in the medium and large models. In order to access data of class *code*, the 8051 compiler supports pointers to *code*.

The most common application of *code* objects is to store strings in ROM. Clearly, such strings cannot be accessed by routines like *printf()* and *puts()* which accept normal *char* * arguments. A routine to write a *code* string in the same manner as *puts()* could be encoded as:

```
void
code_puts(code char * codeptr)
{
        char    ch;
        while (ch = *codeptr++)
                putch(ch);
        putch('\n');
}
```

Code_puts() could then be used to display strings directly from ROM as follows:

```
extern void     code_puts(code char *);
```

```
code char       hello[] = "Hello, world\n";
main()
{
        code_puts(hello);
}
```

Care must be taken to avoid passing pointers to *code* to any routine which expects a default or *far* pointer. Likewise, normal and *far* pointers should not be passed to routines like *code_puts()*. Naturally the compiler will assist with appropriate warning messages if it detects casts between incompatible pointer classes.

11.3.10 Pointer Types

HI-TECH C supports several different classes of pointer, of both 8 and 16 bit size. 8 bit pointers may only access data objects which reside in internal RAM. 16 bit pointers may access be used to access objects in internal RAM, external RAM and the CODE address space depending on the usage and the class of the pointer. The default pointer class, unmodified by any class keywords such as *code*, is a 16 bit pointer which addresses a 64K address space which is the concatenation of internal RAM (00H to FFH) and either the CODE space (in small model), or the external RAM space (in the medium and large models).

11.3.10.1 Pointers in small model

It is important to remember that in small model, pointers to addresses above FFH do not address any RAM. When using small model, if you wish to access external RAM you will need to use the *far* or *xdata* type qualifier both to declare external RAM variables and to declare pointers which can address external RAM. To illustrate the behaviour of pointers in small model, the code:

```
char    ch1, ch2;
main()
{
        char *  ptr;
        ptr = (char *) 0x60;
        ch1 = *ptr;
        ptr = (char *) 0x1060;
        ch2 = *ptr;
}
```

will read internal RAM location 60H into ch1 and ROM location 1060H into ch2. To access external RAM location 1060H a pointer of class *far* would need to be used, for example:

Table 11.5: Pointer classes — small model

Pointer class	Address space	Size	Example
default	IRAM+CODE	16 bit	`char *ptr`
`near`	IRAM	8 bit	`near char *ptr`
`idata`	IRAM	8 bit	`idata char *ptr`
`const`	IRAM+CODE	16 bit	`const char *ptr`
`far`	XDATA	16 bit	`far char *ptr`
`xdata`	XDATA	16 bit	`xdata char *ptr`
`pdata`	XDATA	8 bit	`pdata char *ptr`

```
char    ch2;
main()
{
        far char *      ptr;
        ptr = (far char *) 0x1060;
        ch2 = *ptr;
}
```

Attempts to write to memory via pointers with values above FFH cause undefined results as the 8051 does not have instructions to write to CODE memory. Once again, if you want to write to external RAM in small model, you must use variables and pointers of class *far* or *xdata*. If only 8 bits of the external data bus are decoded then you can also use the *pdata* pointer, which will use `movx a,@r1` type instructions. *Pdata* pointers are 8 bits wide.

The available pointer classes in small model are listed in Table 11.5. All of the pointer classes listed may also be combined with the ANSI C *const* and *volatile* type qualifiers. The *const* type qualifier has the effect of prohibiting indirect writes via pointers, the *volatile* type qualifier disables optimization of apparently redundant accesses and should be used when declaring pointers to memory mapped I/O devices.

11.3.10.2 Pointers in the medium, large and huge models

In the medium, large and huge memory models, the default pointer classes address internal RAM when pointing to addresses 00H to FFH and external RAM when pointing to addresses 0100H to FFFFH. In these models, the default pointer classes suffer from none of the limitations of pointers in small model.

To use the same example shown for small model:

Table 11.6: Pointer classes — medium, large and huge models

Pointer class	Address space	Size	Example
default	IRAM+XDATA	16 bit	`char *ptr`
`near`	IRAM	8 bit	`near char *ptr`
`idata`	IRAM	8 bit	`idata char *ptr`
`const`	IRAM+XDATA	16 bit	`const char *ptr`
`code`	CODE	16 bit	`code char *ptr`
`far`	IRAM+XDATA	16 bit	`far char *ptr`
`xdata`	XDATA	16 bit	`xdata char *ptr`

```
char    ch1, ch2;
main()
{
        char *  ptr;
        ptr = (char *) 0x60;
        ch1 = *ptr;
        ptr = (char *) 0x1060;
        ch2 = *ptr;
}
```

If compiled as either medium, large or huge model code, this example will read the contents of internal RAM location 60H into ch1 and the contents of external RAM location 1060H into ch2. Attempts to write to memory locations above FFH will behave as expected, writing to external RAM. In the medium and large models it is necessary to use the *code* qualifier to access data stored in ROM.

The available pointer classes in the medium, large and huge models are listed in Table 11.6. All of the pointer classes listed may also be combined with the ANSI C *const* and *volatile* type qualifiers. The *const* type qualifier has the effect of prohibiting indirect writes via pointers, the *volatile* type qualifier disables optimization of apparently redundant accesses and should be used when declaring pointers to memory mapped I/O devices.

To access external data memory unconditionally in medium, large, and huge model, use the *xdata* qualifier.

11.3.10.3 Function Pointers

Function pointers can be defined to indirectly call functions or routines in the program space. The size of these pointers are 16 bits wide, with the exception of far functions in huge model which are 32 bits wide. Note that 16 bit function pointers to *near* and *basenear* functions in huge model are currently unsupported. The addresses for all code labels are always shown in the map file as an untruncated byte address regardless of the options used.

11.3.10.4 Combining type modifiers and pointers

The *const*, *volatile*, *idata*, *near*, *far* and *code* modifiers may also be applied to pointers, controlling the behaviour of the object which the pointer addresses. For example, you may declare a pointer to *near*, which is an 8 bit pointer addressing only objects in internal RAM.

When using these modifiers with pointer declarations, care must be taken to avoid confusion as to whether the modifier applies to the pointer, or the object addressed by the pointer. The rule is as follows: if the modifier is to the left of the "*" in the pointer declaration, it applies to the object which the pointer addresses. If the modifier is to the right of the "*", it applies to the pointer variable itself. Using the *near* type qualifier to illustrate, the declaration:

```
near char * nptr;
```

declares a pointer to a *near* character, i.e. a character which is located within internal RAM. The *near* modifier applies to the object which the pointer addresses because it is to the left of the "*" in the pointer declaration.

In small model, the C statement `*nptr = 1;` will generate this code:

```
MOV     R0,_nptr
MOV     @R0,#1
```

The declaration: `char * near ptr;` behaves quite differently however. The *near* qualifier is to the right of the "*" and thus applies to the actual pointer variable *ptr*, not the object which the pointer addresses. This declaration produces a pointer variable which resides in the *near* address space.

Finally, the declaration: `near char * near nnptr;` will generate a pointer variable which resides in internal RAM and which can only address objects in the first 256 bytes of memory. This is the most efficient possible pointer type on the 8051. Note that if you are using small model, all variables are placed in internal RAM unless specifically declared to be *far*. The type qualifiers *idata*, *far*, *code*, *const* and *volatile* may also be applied to pointers.

11.3.10.5 Near and Idata pointers

Pointers to classes *near* and *idata* may be declared by prefixing the "*" in the declaration with *near* or *idata*. For example:

```
near char * near_ptr;
idata char * idata_ptr;
```

declare pointers to class *near* and *idata* respectively.

The positioning of the qualifier to the left of the “*” is not important, thus:

```
near char * near_ptr;
```

and

```
char near * near_ptr;
```

are equivalent declarations. Pointers to *near* and *idata* are both 8 bit pointers which can address the 256 internal RAM area. Note that pointers to addresses 80H to FFH access the indirect internal RAM area, NOT the special function registers at the same addresses. *Near* and *idata* may be mixed with the ANSI standard modifiers *const* and *volatile* to declare variables and pointers which are both *near* or *idata* and *const* or *volatile*.

Variables of class *pointer to near* and *pointer to idata* behave identically in all cases. However, there is one case where *near* and *idata* are not equivalent, that is, the de-referencing of constant pointers.

The C language allows a constant value to be cast to a pointer and then de-referenced. HI-TECH C extends this type casting capability to include all special pointer classes including *near* and *idata*. It is possible to de-reference a constant pointer with a statement like: `ch = *(near char *)0x90;` .

A constant de-reference of a *near* pointer such as the above example will generate code which accesses SFR location 90H, NOT internal RAM location 90H.

On the other hand, the statement: `ch = *(data char *)0x90;` will generate code which accesses internal RAM location 90H, not SFR location 90H.

Put another way, de-referencing a constant *near* pointer generates code using the DIRECT addressing mode of the 8051, such as: `MOV _ch,90H` whereas de-referencing a constant *idata* pointer will generate code which uses the INDIRECT address mode, such as:

```
MOV     R0,#90H
MOV     _ch,@R0
```

Use of constant pointers to access internal RAM or SFR space is strongly discouraged. Direct access to special function registers should be achieved using the absolute variable facility.

11.3.10.6 Far pointers

HI-TECH C allows pointers to class *far* to be declared. Pointers to *far* are 16 bit pointers which can be used to access the 64K external RAM (XDATA) area on the 8051. *Far* pointers are of most use in the small memory model and probably will not be required in medium, large, or huge model code, since a *far* pointer in these models is the same as a default pointer - it accesses internal memory below 100h.

A variable of class *pointer to far* can be declared as follows:

```
far char * far_ptr;
```

The above declaration produces a 16 bit pointer *far_ptr* which can be used to access external RAM. All accesses to **far_ptr* are performed via the `MOVX A,@DPTR` and `MOVX @DPTR,A` instructions of the 8051.

11.3.10.7 Xdata pointers

An *xdata* pointer will always access external data memory, in all models. It will produce movx instructions.

11.3.10.8 Pdata pointers

If small model is used, and only 8 bits of external data address are decode, then the *pdata* qualifier may be useful. A pointer to *pdata* will occupy only 8 bits, and when the pointer is dereferenced a `movx` a,@r1 instruction or similar will be generated.

11.3.10.9 Code pointers

The *code* type qualifier is used to declare constants which are placed in ROM and accessed using the `MOVC A,@A+DPTR` instruction. HI-TECH C allows variables of class *pointer to code* to be declared. Pointers to *code* are of most use in the medium, large, and huge models where the default pointer class addresses external RAM. Small model code will not need to use the *code* qualifier. A common use of *code* constants and variables of class *pointer to code* is to access string constants such as menus and prompts which have been placed in ROM.

The following code illustrates this technique:

```
#include <conio.h>
static code char hello[] = "Hello, world\n";
static void
code_puts(code char * cptr)
{
```

```
		char ch;
		while (ch = *cptr++)
			putch(ch);
	}
	main()
	{
		code_puts(hello);
	}
```

Use of *code* constants and pointers can reduce external RAM usage, particularly in the medium, large, and huge memory models, which copy initialized variables to external RAM.

11.3.10.10 Const pointers

Pointers to *const* should be used when indirectly accessing objects which have been declared using the *const* qualifier. *Const* pointers behave in nearly the same manner as the default pointer class in each memory model, the only difference being that the compiler forbids attempts to write via a pointer to *const*.

Thus, given the declaration:

```
const char *	cptr;
```

the statement: `ch = *cptr;` is legal, but the statement: `*cptr = ch;` is not.

In the small model, const pointers always access program ROM, because const declared objects are stored in ROM. In the other models const pointers behave like normal pointers, except that you may not write to memory via a const pointer. The const class may be combined with the classes near and far to produce variables and pointers to constant objects in either of these address spaces.

Thus the declaration:

```
near const * ncptr;
```

produces a variable *ncptr* which is an 8 bit pointer to *const* characters in internal RAM. It is possible to read internal RAM by de-referencing *ncptr*, but the statement: `*ncptr = 0;` will be rejected by the compiler.

11.4 Storage Class and Object Placement

11.4.1 Local variables

C supports two classes of local variables in functions: *auto* variables which are normally allocated on some sort of stack and *static* variables which are always given a fixed memory location.

11.4.1.1 Auto Variables

Auto variables are the default type of local variable. Unless explicitly declared to be *static* a local variable will be made *auto*.

Due to architectural limitations on the 8051, in the small and medium models *auto* variables are not allocated on the stack and are instead given fixed addresses within the *rbss* psect. The name of the *auto* variable block for a function will be the name of the function with **?a_** prepended.

For example, the following function:

```
void test(void)
{
        int     i;
        char    c, k;
        i = 10;
        c = 20;
        k = 30;
}
```

has 4 bytes of *auto* variables in a block called *?a_test*, as is illustrated by the code generated for the three assignments:

```
;x.c: 6: i = 1;
        MOV     ?a_test,#0
        MOV     ?a_test+1,#10
;x.c: 7: c = 2;
        MOV     ?a_test+2,#20
;x.c: 8: k = 3;
        MOV     ?a_test+3,#30
```

Auto variables may be overlaid by storage allocated for other functions. The 8051 does have an addressable stack, but the linker will allocate the same local variable addresses to functions which can never be active at the same time. *Auto* variables are not guaranteed to retain their value between successive calls to a function.

A function in small or medium model may be declared to be *reentrant* in which case arguments and auto variables will be stored on the 8051 stack. This allows a function to be called re-entrantly or recursively.

In the large and huge models, *auto* variables are allocated on the external stack and accessed via the MOVX instruction. These models permit fully re-entrant and recursive code; any function may be invoked more than once without corruption of function arguments and *auto* variables. If global optimization is used, some *auto* variables may be placed in registers.

11.4.1.2 Static Variables

Static variables are allocated in the *bss* psect and occupy fixed memory locations which will not be overlapped by storage for other functions. *Static* variables are local in the function which they are declared in, but may be accessed by other functions via pointers. *Static* variables are guaranteed to retain their value between calls to a function, unless explicitly modified via a pointer. *Static* variables are not subject to any architectural limitations on the 8051.

11.4.2 Absolute Variables

A global or static variable can be located at an absolute address by following its declaration with the construct @ *address*, for example:

```
volatile unsigned char P1 @ 0x90;
```

will declare a variable called P1 located at 90H in the special function register area of the 8051 address space.

Note that the compiler does not reserve any storage, but merely equates the variable to that address, the compiler generated assembler will include a line of the form:

```
_P1 equ 90h
```

Absolute variables provide a convenient method of accessing the built in special function registers of the 8051. Absolute variable declarations may be combined with the *far* type qualifier to access memory mapped I/O devices in the external address space.

For example:

```
volatile far unsigned char   SIO_A_DATA @ 0x1FF0;
```

will declare a variable called *SIO_A_DATA* located at 1FF0H in the XDATA address space. This location will be accessed using the 8051 *MOVX* instruction.

For example the C statement

```
SIO_A_DATA = 'A';
```

will produce these 8051 instructions:

```
MOV     DPTR,#1FF0H
MOV     A,#65
MOVX    @DPTR,A
```

11.5 Functions

11.5.1 Function Argument passing

Although the 8051 processor does have an addressable stack, the small and medium memory models only use the stack to store function return addresses. A combination of register based argument passing and static memory allocation is used for function arguments.

Auto variables are allocated to static locations or registers in the small and medium models. If global optimization is used, registers may be used to hold some *auto* variables and arguments. The large and huge memory models use an external stack for *auto* variables and some argument passing.

The large and huge models will also use registers to pass arguments, in the same manner as the small and medium models. If global optimization is used, registers may be used to hold some *auto* variables and arguments.

11.5.1.1 Small and medium model argument passing

The small and medium models use the same scheme for function arguments, a combination of register and static memory based argument passing. Generally function arguments are passed in static memory locations in internal RAM (except in *reentrant* functions), with the left most argument at the lowest address. The name of the argument block for a function is the name of the function with the character *?_* prepended. Thus the arguments for a function called *test()* will be passed in a block of internal memory called *?_test*. The function argument block will be allocated in the *rbss* psects and such arguments will be accessed in the same manner as any other internal memory variable.

In addition to the static argument scheme detailed above, HI-TECH C will pass up to 4 bytes of function arguments in registers R2, R3, R4 and R5. Register based argument passing only occurs with functions which have an ANSI C style function prototype. Functions which use old style C declarations will receive all arguments in static memory locations. For functions which have an ANSI style prototype, some arguments will be passed in registers R2, R3, R4 and R5.

The rules for register based argument passing are as follows:

- Only the left most two arguments to a function will be passed in registers. All other arguments will be passed in static memory locations.
- Only 8 bit and 16 bit arguments will be passed in registers. 32 bit arguments and structures of size 24 bits and larger will be passed in static memory locations.
- Any argument followed by a variable argument list (...) will be passed in static memory locations.
- If the first argument to a function is an 8 bit quantity, it will be passed in register R5.

- If the first argument to a function is a 16 bit quantity, it will be passed in registers R4 and R5 with the high order byte in R4 and the low order byte in R5. Structures and unions of size 16 bits will also be passed in R4 and R5.
- If the second argument to a function is an 8 bit quantity, it will be passed in register R3.
- If the second argument to a function is a 16 bit quantity, it will be passed in registers R2 and R3 with the high order byte in R2 and the low order byte in R3. Structures and unions of size 16 bits will also be passed in R2 and R3.

The following examples demonstrate the argument passing mechanisms used by the small and medium memory models:
void char_func(char ch);
will receive argument *ch* in register R5. For example, the call char_func(10) will generate the code:

```
MOV     R5,#10
LCALL   _char_func
```

void int_func(int i);
will receive argument **i** in register R4 and R5. The C statement:

```
int_func(0x1234);
```

will call *int_func()* with 12H in R4 and 34H in R5.

```
void long_func(long l);
```

will receive argument **l** in a 4 byte block of internal RAM called *?_long_func*. The call:

```
long_func(0x12345678);
```

will generate this code:

```
MOV     ?_long_func,#12H
MOV     ?_long_func+1,#34H
MOV     ?_long_func+2,#56H
MOV     ?_long_func+3,#78H
LCALL   _long_func
```

The call:

```
void var_args(char * str, ...);
```

will receive argument str in locations *?_var_args* and *?_var_args+1* because *str* is followed by a variable argument list. See the manual Section 11.5.1.5 for a discussion of variable argument list passing in the small and medium models.

```
void multi_args(long l, int i1, int i2);
```

will receive arguments *l* and *i2* in static memory locations and *i1* in R2 and R3. The call:

```
multi_args(1, 2, 3);
```

will generate this code:

```
MOV     ?_multi_args+4,#0
MOV     ?_multi_args+5,#3
MOV     R2,#0
MOV     R3,#2
MOV     ?_multi_args,#0
MOV     ?_multi_args+1,#0
MOV     ?_multi_args+2,#0
MOV     ?_multi_args+3,#1
LCALL   _multi_args
```

11.5.1.2 Reentrant functions

In small and medium model it is possible to declare a function to be *reentrant*, which will have the effect of allocating auto variables and parameters on the 8051 stack, instead of statically in memory. This will mean the function can be called re-entrantly or recursively. The keyword is simply inserted before the function name e.g.

```
char * reentrant a_func(int arg)
{
/* function body here */
}
```

11.5.1.3 Large and huge model argument passing

The large and huge memory models use a combination of register and external stack based argument passing. The rules for register based argument passing are the same as in the small and medium memory models, as are the registers used. External stack based arguments are pushed onto a downward growing stack using library routines. The calling function is responsible for both pushing and

removing the arguments. In order to minimize stack usage in these models, the function return address is saved on the external stack on entry, and restored on exit. This allows functions to be called in a fully re-entrant and recursive manner, limited only by the amount of external RAM available for the stack.

11.5.1.4 Variable argument lists

The *small* and *medium* models pass variable argument lists by storing any unnamed arguments into a local variable block, belonging to the caller and passing the address of the variable argument block in the accumulator. This scheme allows variable argument lists to be passed with the same efficiency as normal arguments. Variable argument lists work in the normal manner in the large and huge memory models, each argument is pushed onto the stack in right to left order resulting in the argument list appearing in the correct order in memory.

11.5.1.5 Small and medium model variable argument lists

Each function which calls another function using a variable argument list will use extra local variable space equal in size to the largest variable argument list passed within that function.

For example, if *main()* calls *printf()* twice, with 4 bytes of variable arguments for the first call and 10 bytes of variable arguments for the second call, the local variable area *?_main* will include 10 bytes for the variable argument block. If a call to a different function using 8 bytes of variable arguments were added, the variable argument area would not be enlarged.

To illustrate the behaviour of variable argument lists, the following code:

```
extern void     printf(char *, ...);
int     var;
char *  name;
char *  format = "%s = %d\n";
main()
{
        printf(format, name, var);
}
```

produces this code when compiled:

```
MOV     ?a_main+2,_var
MOV     ?a_main+3,_var+1
MOV     ?a_main,_name
MOV     ?a_main+1,_name+1
MOV     ?_printf,#high _format
```

```
MOV     ?_printf+1,#low _format
MOV     A,#?a_main
LCALL   _printf
```

11.5.1.6 Indirect function calls

HI-TECH C fully supports the use of function pointers to indirectly call functions. In the large and huge models, indirect functions occur in the normal C manner with arguments passed in registers and on the stack.

11.5.1.7 Small and medium model indirect function calls

In order for indirect calls to functions which take memory based arguments to work in these models, the compiled code needs to be able to locate where the static argument block for a particular function resides. Two functions which have the same prototype but different argument addresses may both be called via the same function pointer. This problem is overcome by embedding the argument block address in the code, one byte before the start of the function. When performing an indirect function call the code will be able to find the argument block address by looking one byte before the address specified by the pointer.

For example, the function:

```
long
add_10(long arg1)
{
        return arg1 + 10;
}
```

will generate this code:

```
        DB      ?_add_10
_add_10:
        MOV     A,?_add_10+3
        ADD     A,#10
        MOV     R5,A
        MOV     A,?_add_10+2
        ADDC    A,#0
        MOV     R4,A
        MOV     A,?_add_10+1
        ADDC    A,#0
        MOV     R3,A
```

```
MOV     A,?_add_10
ADDC    A,#0
MOV     R2,A
RET
```

The byte just before label *_add_10* points at the argument block for the function *?_add_10*. *Add_10()* could be indirectly accessed by a function pointer such as:

```
long    (*funcptr)(long);
```

The C statement:

```
res = (*funcptr)(value);
```

will generate this code:

```
MOV     A,_funcptr+1
ADD     A,#255
MOV     DPL,A
MOV     A,_funcptr
ADDC    A,#255
MOV     DPH,A
CLR     A
MOVC    A,@A+DPTR
MOV     R1,A
MOV     @R1,_value
INC     R1
MOV     @R1,_value+1
INC     R1
MOV     @R1,_value+2
INC     R1
MOV     @R1,_value+3
MOV     R0,_funcptr
MOV     R1,_funcptr+1
LCALL   indir
MOV     _res,R2
MOV     _res+1,R3
MOV     _res+2,R4
MOV     _res+3,R5
```

Note the use of the library routine *indir* to call the function indirectly. The code above loads the argument block address into R1 and then uses indirect addressing to store the arguments into the correct area in memory for the function which is to be called.

11.5.2 Function return values

Function return values are passed to the calling function as follows:

11.5.2.1 8 Bit return values

8 bit values (*Char*, *near pointer* and *idata pointer*) are returned in register R3. For example, the C function:

```
char
return_zero(void)
{
        return 0;
}
```

will exit with the following code:

```
        MOV     R3,#0
        RET
```

11.5.2.2 16 Bit return values

16 bit values (*Int*, *short* and *pointer*) are returned in the R2 and R3 registers with the least significant byte in R3 and the most significant byte in R2. Thus the following function:

```
int test(void)
{
return 0x1234;
}
```

will return with 0x34 in R3 and 0x12 in R2.

11.5.2.3 32 Bit return values

32 bit values (*long* and *float*) are returned in registers R2, R3, R4 and R5 with the most significant byte in R2 and the least significant byte in R5. This is illustrated by the following code:

```
long return_long(void)
{
        return 0x01020304;
}
```

which will exit using the sequence of instructions:

```
MOV     R2,#1
MOV     R3,#2
MOV     R4,#3
MOV     R5,#4
RET
```

11.5.2.4 Structure return values

Composite return values (*struct* and *union*) are returned by various means depending on size. 8 bit structures are returned in register R3, 16 bit structures in R2 and R3 and 32 bit structures in R2, R3, R4 and R5. Large structures are returned by reference and copied by calling a library routine called str_copy.

11.5.3 Function Calling Conventions for Huge Model

When using the huge (bankswitched) model, the calling conventions are similar to large model except for the actual call. Rather than calling the function directly, register B is loaded with the bank number of the function to be called, DPTR is loaded with its address within that bank, and then a call is made to the *bcall* routine in common memory which performs the necessary bank switching before jumping to the function. The current bank is saved on the internal stack. On return from the function, a jump to the *bret* routine is performed which retrieves the old bank number from the stack before a return is made to the calling function. This implies a maximum restriction on nested banked subroutine calls, depending on internal stack usage for interrupts, temporary usage and so forth. Functions qualified as *near* or *basenear* are not affected by this restriction.

To implement a custom banking scheme, a replacement *bcall* module must be implemented. See the *bcall.as* file in the SOURCES directory of the compiler for code used in the standard libraries. This file includes cases when a bank select register mapped into both internal and external memory. Care must be taken when implementing a custom *bcall* module to preserve all registers apart from DPTR, B, and the accumulator.

11.5.3.1 Near and Basenear Functions in Huge Model

When using the *huge* (i.e. banked) model, functions are called using the mechanism described above by default. It is, however, possible to define functions that are called via a simple *lcall/ret* sequence, thus speeding up the code. The two ways to do this are with *near* functions and with *basenear* functions.

A *near* function can be called only from within the same bank, while a *basenear* function resides in the common area and can be called from any bank. Near functions should be declared static and

called and called only from within the same module. The following code shows an example of these kind of functions.

```
static near int
read(void)
{
            while(!RI)
                  ;
            return SBUF;
}

basenear void
reset( void)
{
            RI = 0;
}
```

11.5.4 The call graph

In order to preserve memory, the linker performs stack like allocation of function arguments and local variables using a technique called “call graphing”. Call graph analysis allows the linker to determine which functions call and are called by other functions and build a graph of dependencies. The linker will analyse the call graph and determine which functions can never be active at the same time, making it safe to overlap their local variable and argument areas.

For example, consider the following C code:

```
void func_a(int arg1)
{
        int     var1, var2;
.
.
.
}
void func_b(int arg)
{
        long    l_var;
.
.
.
}
```

```
main()
{
        func_a(1, 2);
        func_b(0x1000);
}
```

main() calls both *func_a()* and *func_b()*, so the variable block for *main()* cannot occupy the same memory as the variable blocks for either function. *Func_a()* is never called at the same time as *func_b()*, so it is safe for the local variables and arguments belonging to both functions to occupy the same memory, exactly as would occur if a stack were used for variable storage. The call graphing technique makes it possible to write code containing a large number of functions and local variables without worrying too much about using all of the available internal RAM on the 8051. Even if there are 100 functions with 10 bytes of local variables each, if none of the functions are ever active at the same time only 10 bytes of local variable space will be used.

11.6 Memory Models and Usage

The compiler makes few assumptions about memory. With the exception of variables declared using the *@address* construct, absolute addresses are not allocated until link time. Certain classes of variable are assumed to reside within particular address ranges and address spaces, as limited by the 8051 architecture.

The memory used is based upon information in the chipinfo file (which defaults to `8051.ini` in the LIB directory). The linker will automatically locate code and `const`-qualified data into all the available memory pages and ensure that psects do not straddle any memory boundary.

There are four memory models available for C51: small, medium, large, and huge, the default of which is small. The memory model is selected via the `-B`*x* command line option. See Section 10.4.1.

Small model is a fully static model which does not support re-entrant or recursive code. *Extern*, *static* and *auto* variables, and function arguments, are allocated statically in internal RAM. *Extern* and *static* variables may be allocated in external RAM using the *far* qualifier.

Medium model is also a fully static model which does not support re-entrant or recursive code. However, *extern* and *static* variables are allocated in external RAM, *auto* variables and function arguments are allocated statically in internal RAM. *Extern* and *static* variables may be allocated in internal RAM using the *near* and *idata* qualifiers.

In both small and medium models, call graphing is used by the linker to overlay *auto* variables and arguments of functions which can never be active at the same time.

Large model is a fully re-entrant code generation model which uses a downward growing stack in external RAM to bypass the 8 bit stack pointer limit imposed by the 8051, the top address of which is calculated from the highest usable address in external RAM. *Extern*, *static* and *auto* variables are

all allocated in external RAM, *auto* variables and function arguments are allocated on the external stack. *Extern* and *static* variables may be placed in internal RAM using the *near* and *idata* qualifiers.

Huge model is equivalent to large model, with the added functionality of utilising a banked code configuration. When using huge model, functions are by default qualified *far*. This places them into the banked region in the *ltext* psect (See Section 11.8) filling additional banks as required. Functions may be placed in the common region by using the *basenear* qualifier. See Section 11.5.3.

Function addresses in huge model are 24 bits, but 32 bits is actually allocated where a function pointer is stored in memory. A limitation on the levels of nested banked calls exists due to the storage of the segment number (bits 16-24 of the function address) on the internal stack.

11.7 Register usage

With two exceptions, compiled code always assumes that register bank 0 is selected. The exceptions are code within *bank2 interrupt* functions which assume that register bank 2 is selected, and *bank3 interrupt* functions which assume that register bank 3 is selected.

Some library routines use register bank 1, but restore register select bits on return to re-select bank 0.

Registers R0 and R1 are used as temporary values and for indirectly addressing data in internal RAM. All accesses to variables of class *idata* will make use of either R0 or R1, as will de-references of *near* pointers and standard points with values less than 100H.

Registers R2, R3, R4 and R5 are used for register based argument passing and for function return values. These registers will also be used to hold temporary values within functions and may also be used to contain arguments or local variables if code is compiled with global optimization.

Registers R6 and R7 are used to hold register variables in the small and medium models, and for the external stack pointer in the large and huge memory models. R6 and R7 should be preserved by any assembly language routines which are called.

The accumulator (ACC) and B register are used for arithmetic operations and as a scratch pad. B is used as an operand to the MUL and DIV, and in huge model, is used to load the segment selector for a banked subroutine call.

The PSW register varies depending on the operation of user code, however the register bank select bits should be preserved by assembly language routines.

The DPTR register (DPL and DPH) is used as a scratch pad, and for pointer operations which access external RAM or program ROM.

11.8 Compiler generated psects

The compiler splits code and data objects into a number of standard program sections, referred to as psects. The HI-TECH assembler allows an arbitrary number of named psects to be included in

assembler code. The linker groups all data for a particular psect into a single segment.

If you are using C51 to invoke the linker, you don't need to worry about the information documented here, except as background knowledge.

If you want to run the linker manually, or write your own assembly language subroutines you should read this section carefully.

The psects used by compiler generated code are:

vectors The *vectors* psect contains the reset vector followed by all initialized interrupt vectors. *Vectors* is normally linked for address 0 in ROM so that the LJMP start instruction at the beginning of the psect aligns with the 8051 reset vector.

text is used for all executable code. By default the C compiler places all executable code in the *text* psect (with the exception of huge model, see *ltext* below). User written assembly language subroutines should also be placed in the *text* psect.

ltext In huge model only, the *ltext* psect contains all executable code to be placed in the banked region. The linker will automatically fill additional banks as required. This is the default psect for executable code in huge model unless functions are qualified *basenear*.

code is used for any statically initialized constants of class *code*. For example:`code int maxdata = 10;` declares a constant *maxdata* with value 10 which resides in the *code* psect. *Code* is linked into program ROM after the *text* psect, objects in the *code* psect are accessed using the MOVC instruction.

const is used for all initialized constants of class *const*, for example:`const char masks[] = { 1,2,4,8,16,32,64,128} ;`

strings The *strings* psect is used for all unnamed string constants, such as string constants passed as arguments to routines like *printf()* and *puts()*.

data The *data* psect is used to contain all statically initialized data except those in classes *near*, *code* and *const*. For the small memory model, the *data* psect is linked into ROM, statically initialized data items are not modifiable and are accessed using the MOVC instruction.

For the medium, large and huge memory models, the *data* psect is linked into external RAM, with a copy in ROM (placed in the *zdata* psect) which is transferred to external RAM by the run-time startup code. Statically initialized data items may be modified like any other variable in these models.

zconst In the medium, large, and huge models, the *zconst* psect contains the ROM image of any initialized constants which are copied into the *const* psect at startup.

zstrings In the medium, large, and huge models, the *zstrings* psect contains the ROM image of any unnamed string constants which are copied into the *strings* psect at startup.

zdata In the medium, large, and huge models, the *zdata* psect contains the ROM image of any statically initialized data (except those in classes *near*, *code* and *const*), which are copied into the *data* psect at startup.

rdata contains all statically initialized variables of class *near*, for example:`static near int size = 256;`
The rdata psect behaves in the same manner for all memory models. Initialized data will only be placed in rdata if declared to be near, otherwise it will be placed in the data psect.
A copy of the rdata psect is stored in ROM and transferred to internal RAM by the run-time startup code before *main()* is invoked.

irdata contains all statically initialized variables of class *idata*, for example:`idata int isize = 128;`

The *irdata* psect behaves in the same manner for all memory models. Initialized data will only be placed in *irdata* if declared to be *idata*. A copy of the irdata psect is stored in ROM and transferred to internal RAM by the startup code before *main()* is invoked. *Irdata* objects may reside at internal addresses above 7FH and are always accessed via register indirect addressing.

bss The *bss* psect is used for all uninitialized static and extern variables which reside in external RAM. *Bss* is cleared to all zeros by the run-time startup code before *main()* is invoked.
For the small memory model, *bss* only contains variables which have been declared as *far*.
For the medium, large and huge models the *bss* psect contains all uninitialized static and extern variables except those which have been declared to be of class near or idata.

rbss contains any uninitialized variables of class *near*. The *rbss* psect is linked into internal RAM at addresses below 7FH, and is accessed using the direct addressing mode of the 8051. This psect is cleared to all zeros by the run-time startup code before *main()* is invoked.
The actual classes of variable which go in *rbss* depend on which memory model is being used.
In the small memory model, *rbss* will be used for all *near* variables, *auto* variables, function arguments and any static and extern variables which are not declared to be in any other class.
In the medium memory model, *rbss* will be used only for *near* variables, *auto* variables and function arguments.
In the large and huge memory models, *rbss* will only be used for variables declared as *near*.

idata contains any variables of class *idata*, for example:`static idata unsigned char counter;` declares *counter* to be of class *idata*. Variables in the *idata* psect are always accessed using the register indirect addressing mode of the 8051. Thus, the C statement ++counter will generate this code:`MOV R0,#_counter`

INC @R0
Idata class variables may reside at addresses in the range 80H to FFH, allowing the “hidden” internal RAM on the 8052, 80C552 and 80C517 to be accessed.

rbit contains all *bit* variables except those declared at absolute locations. The declaration:`static bit unsigned char flag;`
will allocated *flag* as a single bit in the *rbit* psect. The *rbit* psect is always linked for bit addresses in the range 0 to 7FH. Bit addresses 80H to FFH are in the 8051 special function register area and should not be used for the *rbit* psect.

11.9 Using memory mapped I/O and SFRs

The 8051 processor uses memory mapped I/O for all devices. In order to declare memory mapped I/O ports you should use the *absolute* variable facility to map identifiers onto the appropriate special function register locations. The *volatile* type qualifier should be used for most I/O locations to prevent the optimizer from removing apparently redundant reads and writes to ports. When written to, read only I/O ports which do not perform any sensible function, should be declared *const* so that the compiler will detect any attempt to write to them.

Almost all I/O locations on the 8051 family are 8 bits wide, so you should use the *unsigned char* type in your port declarations. The *unsigned char* type is guaranteed to be an 8 bit wide unsigned integer regardless of the compiler options used. The default behaviour of the *char* type is signed, but will behave like an 8 bit unsigned integer if the C51 option –char=unsigned is used.

To give a practical example, the on-board serial port on the 8051 could be declared as follows:

```
static unsigned char    SCON @ 0x98;
static unsigned char    SBUF @ 0x99;
```

Any of the I/O ports declared above could then be used freely in C code, exactly like any other C variable. For example, to write the character ’X’ to the serial port buffer SBUF use the C statement:

```
SBUF = 'X';
```

Declarations for all of the standard 8051 ports may be found in the standard header file <8051.h>. See the appropriate processor handbook for documentation of the special function registers on your hardware.

11.10 Interrupt handling in C

The compiler incorporates features allowing interrupts to be handled without writing any assembler code. The type qualifier *interrupt* may be applied to a function to allow it to be called directly from

a hardware or software interrupt. The compiler will process *interrupt* functions differently to normal functions, generating code to save and restore any registers used and exit using a *RETI* instead of a *RET* at the end of the function. If the C51 option -STRICT is used, this keyword becomes *__interrupt*. Wherever this manual refers to the *interrupt* keyword, assume *__interrupt* if you are using –STRICT.

An *interrupt* function must be declared as type *interrupt void* and may not have parameters. It may not be called directly from C code, but it may call other functions itself, subject to certain limitations.

In the small and medium memory models, static locations are used for *auto* variables and function argument passing. As a result of the static allocation scheme, *interrupt* functions may not make a function call to any function which uses static memory variables, and which is also called from the main program or by a different *interrupt* function. This limitation is imposed because such a call may result in corruption of variables and arguments if another instance of the function is already active. *Interrupt* functions may call any function, with any number of arguments of local variables, if the function called is not used by any other part of the program. In the large model, interrupt functions reserve 256 bytes of memory for their own stack space. At present, the size of this stack space is fixed.

If the linker detects a function call which breaks these rules, it will issue the warning "*Function name* occurs in multiple call graphs, rooted at NAME1 and NAME2" where NAME1 and NAME2 are the name of the *interrupt* function and the name of your *main()* function or another *interrupt* function. Functions which use no static storage may be freely called by any number of *interrupt* and standard functions.

An example an *interrupt* function which services the standard on-board serial port of the 8051 follows:

```
char  rxbuf[16];
volatile char  head, tail;
interrupt void
serial_intr(void)
{
        rxbuf[head] = SBUF;
        head = (head + 1) % sizeof(rxbuf);
        if (head == tail)
                tail = (tail + 1) % sizeof(rxbuf);
        RI = 0;
}
```

11.10.1 Bank2 and *Bank3* interrupts

HI-TECH C supports two special classes of *interrupt* function which switch to register bank 2 or 3 before executing any user code. This saves some processing time compared to standard interrupt functions which push any registers used onto the stack. The keywords *bank2* and *bank3* are used to access this facility. You may declare an interrupt function which uses register bank 2 as follows:

```
bank2 interrupt void func(void);
```

Similarly, an interrupt function using register bank 3 could be declared as:

```
bank3 interrupt void func(void);
```

If the C51 option –STRICT is used, these keywords are changed to *__bank2* and *__bank3*. Interrupt functions using bank 2 or 3 do not generate code to save the registers used, thus reducing the interrupt overhead substantially. Due to the register bank dependant nature of most compiler generated code, banked interrupt functions may not call any other C function. You should not allow more than one interrupt function using the same alternate register bank to be active at a time. You can have as many standard interrupt functions as you like, limited only by the available stack space.

In general, interrupt functions using banks 2 or 3 should be used only to handle interrupts where a very fast response, requiring minimal processing, is desired. For example, an interrupt handler for a timer generating interrupts at a fast rate may be better handled by a bank2 interrupt or bank3 interrupt function.

It is possible to write interrupt handlers which are actually slower when compiled as a banked interrupt function. If the code generated for an interrupt function does not use any of the registers R0 to R7, the code generated to save and restore the register bank will actually make a banked interrupt function larger and slower than a standard interrupt function. If in doubt, compile your C code to an assembly language source file and examine the code which has been generated by the compiler.

11.10.2 Interrupt Levels in small and medium model

Normally it is assumed by the compiler that any interrupt may occur at any time, and an error will be issued by the linker if a function appears to be called by an `interrupt` function and by main-line code, or another interrupt. Since it is often possible for the user to guarantee this will not happen for a specific routine, the compiler supports an interrupt level feature to suppress the errors generated.

This is achieved with the `#pragma interrupt_level` directive. There are two interrupt levels available, and any `interrupt` functions at the same level will be assumed by the compiler to be mutually exclusive. This exclusion must be guaranteed by the user, i.e. the compiler is not able to control interrupt priorities. Each `interrupt` function may be assigned a single level, either 0 or 1.

In addition, any non-`interrupt` functions that are called from an `interrupt` function and also from main-line code may also use the `#pragma interrupt_level` directive to specify that they

will never be called by interrupts of one or more levels. This will prevent linker from issuing an error message because the function was included in more than one call graph. Note that it is entirely up to the user to ensure that the function is *not* called by both main-line and interrupt code at the same time. This will normally be ensured by disabling interrupts before calling the function. It is not sufficient to disable interrupts inside the function after it has been called.

An example of using the interrupt levels is given below. Note that the `#pragma` directive applies to only the immediately following function. Multiple `#pragma interrupt_level` directives may precede a non-interrupt function to specify that it will be protected from multiple interrupt levels.

```
/* non-interrupt function called by interrupt and main-line code */
#pragma interrupt_level 1
void bill(){
inti;
 i = 23;
}

/* two interrupt functions calling the same non-interrupt function */
#pragma interrupt_level 1

void interrupt fred(void)
{
 bill();
}

#pragma interrupt_level 1
void interrupt joh()
{
bill();
}

main()
{
bill();
}
```

Both the low- and high-priority `interrupt` functions may use the interrupt level feature.

Table 11.7: Interrupt handling macros

Macro	Purpose
di()	Disable interrupts
ei()	enable interrupts
ROM_VECTOR	Set up “hard” interrupt vector
set_vector	Setup “hard” interrupt vector
RAM_VECTOR	Setup “soft” interrupt vector
CHANGE_VECTOR	Modify “soft” interrupt vector
READ_RAM_VECTOR	Read a “soft” interrupt vector

11.10.3 Interrupt handling macros

The standard header file *<intrpt.h>* contains several macros which are useful when handling interrupts using C code. These are listed in Table 11.7.

11.10.4 The ei() and di() macros

The *di()* and *ei()* macros may be used to disable and enable maskable interrupts. It may useful to disable interrupts while initializing or servicing I/O devices.

Di() disables interrupts by clearing the EA flag in the ICON special function register using the instruction CLR EA. Similarly, *ei()* enables interrupts by setting EA with the instruction SETB EA.

The 8051 global interrupt enable flag is bit addressable and may accessed from C code using the following *bit* variable declaration:

```
static bit unsigned char   EA @ 0xAF;
```

The declaration above makes it possible to test the interrupt enable state, enable and disable interrupts using C statements. For example, to enable interrupts:

```
EA = 1;
```

11.10.5 ROM_VECTOR and set_vector

ROM_VECTOR is used to set up a “hard coded” vector in ROM which points an 8051 LJMP instruction directly to an interrupt handler. It takes the form:

```
ROM_VECTOR(itt vector, itt handler)
```

where *vector* is the address of the interrupt vector and *handler* is the name of the interrupt function which will be used.

For example, to set the serial interrupt vector at address 23h to point to an interrupt function called *serial_intr()* you could write:

```
ROM_VECTOR(0x23, serial_intr)
```

ROM_VECTOR does not generate any code which is executed at run-time, so it can be placed anywhere in your code. To continue the example above, ROM_VECTOR would have generated the following code:

```
GLOBAL  _serial_intr
PSECT   vectors,ovrld
ORG     0x23
LJMP    _serial_intr
PSECT   text
```

This results in the instruction `LJMP _serial_intr` being placed at offset 23H in the *vectors* psect.

ROM_VECTOR generates in-line assembler code, so the vector address passed to it may be in any format acceptable to the assembler. Hexadecimal interrupt vector addresses may be passed either as C style hex (0x23) or as assembler style hex (23H).

Set_vector is equivalent to *ROM_VECTOR* and is present only for compatibility with version 5 and 6 HI-TECH compilers. It is suggested that *ROM_VECTOR* be used in place of *set_vector* for maximum compatibility with future versions of HI-TECH C.

11.10.6 RAM based interrupt vectors

HI-TECH C supports internal RAM based interrupt vectors which can be dynamically modified by user code, so as to point to different interrupt handlers at different points during program execution.

RAM based interrupt vectors work by setting the ROM based interrupt vector to point to code which transfers control to the actual interrupt handler via an internal RAM based pointer. The transfer of control to the user specified interrupt handler can be achieved with minimal overhead by PUSHing the handler address onto the stack and then executing a RET instruction.

The *RAM_VECTOR*, *CHANGE_VECTOR* and *READ_RAM_VECTOR* macros are used to initialize, modify and read interrupt vectors which are directed through internal RAM based interrupt vectors in the *rdata* psect. These macros should only be used for vectors which need to be modifiable, so as to point at different *interrupt* functions at different points in the program. The CHANGE_VECTOR and READ_RAM_VECTOR macros should only be used with interrupt vectors which have been initialized using RAM_VECTOR, otherwise strange things will happen.

11.10.7 RAM_VECTOR

The RAM_VECTOR macro sets up a "soft" interrupt vector which can be modified to point to a different *interrupt* function if necessary. This is accomplished by setting up code at the vector in ROM to perform an indirect jump to the interrupt function, via a vector address in internal RAM. When the interrupt occurs, the code at the interrupt vector uses two PUSH instruction to place the address of the handler on the stack, then executes a RET instruction to jump to the handler address which has just been pushed. If the interrupt vector needs to be changed, the address operand of the PUSH instruction at the vector points to the "soft" vector which is in internal RAM.

RAM_VECTOR takes the same arguments as ROM_VECTOR and can be used anywhere ROM_VECTOR is used. Each use of RAM_VECTOR results in an extra two bytes of initialized data in the *rdata* psect. For example, the code:

```
RAM_VECTOR(0x23, serial_intr)
```

will place code at interrupt vector 23h which indirectly jumps to the actual interrupt handler.

The internal RAM locations used for the vector will be initialized to contain the address of the interrupt function *serial_intr()*. The code generated will be:

```
GLOBAL  _serial_intr
PSECT   vectors,ovrld
ORG     0x23
PUSH    999f+1
PUSH    999f
RET
PSECT   rdata,class=DATA
999:DW      _serial_intr
PSECT   text
```

This results in the code at vector 23h pushing the address of *serial_intr()* onto the stack and then jumping to it via a RET instruction.

11.10.8 CHANGE_VECTOR

The CHANGE_VECTOR macro is used to modify a vector which has been set up by RAM_VECTOR. This is accomplished by modifying the interrupt handler address in internal RAM. For example:

```
EA = 0;
CHANGE_VECTOR(23h, new_handler)
EA = 1;
```

will change the handler address used by vector 23H to point to an interrupt function called *new_handler()*.

The address of the vector word in internal RAM is found by indirecting from the operand byte of the second PUSH instruction at the vector; if the code at the vector is:

```
PUSH    45H
PUSH    44H
RET
```

CHANGE_VECTOR will place the new handler address at 44H and 45H in internal RAM, with the high order byte of the address at 44H.

It is a good idea to disable interrupts before using the CHANGE_VECTOR macro, as it is possible for an interrupt to be generated while the RAM based interrupt vector is in an inconsistent state. The 8051 is a byte oriented machine, so the two bytes of the handler address are updated by separate instructions.

If a vector has been modified and you want to change it back to the original value, you will need to use CHANGE_VECTOR to change it back. Re-executing the code which contains the RAM_VECTOR macro will not reset the vector because RAM_VECTOR statically initializes the vector without generating any executable code. CHANGE_VECTOR is the only vector initialization macro which generates instructions which are actually executed at run-time, ROM_VECTOR and RAM_VECTOR just force initial values into the vectors.

11.10.9 READ_RAM_VECTOR

The READ_RAM_VECTOR macro may be used to read the value of a RAM based interrupt vector which has been set up by RAM_VECTOR. It must never be used on vectors which have been initialized using ROM_VECTOR as garbage will be returned. READ_RAM_VECTOR can be used along with CHANGE_VECTOR to preserve an old interrupt handler address, set a new address and then restore the original address. For example:

```
volatile unsigned char  wait_flag;
interrupt void
wait_handler(void)
{
        ++wait_flag;
        RI = 0;
}
void
wait_for_serial_intr(void)
{
        interrupt void    (*old_handler)(void);
```

```
        EA = 0;
        old_handler = READ_RAM_VECTOR(23H);
        wait_flag = 0;
        CHANGE_VECTOR(23H, wait_handler);
        EA = 1;
        while (wait_flag == 0)
                continue;
        EA = 0;
        CHANGE_VECTOR(23H, old_handler);
        EA = 1;
}
```

11.10.10 Pre-defined interrupt vector names

The header file <*8051.h*> includes declarations for all of the standard 8051 interrupt vectors. These vector names may be used as the vector address argument to the ROM_VECTOR, set_vector, RAM_VECTOR, CHANGE_VECTOR and READ_RAM_VECTOR macros.

The interrupt vectors defined in <*8051.h*> are listed in Table 11.8. Interrupt vectors other than those in <*8051.h*> may be declared using pre-processor `#define` directives, or the vector address may be directly used with the vector macros.

For example the extra interrupt vectors on the 80C552 microcontroller could be declared as follows:

```
#define I2CINT  0x2B
#define CAP0INT 0x33
#define CAP1INT 0x3B
#define CAP2INT 0x43
#define CAP3INT 0x4B
#define ADCINT  0x53
#define CMP0INT 0x5B
#define CMP1INT 0x63
#define CMP2INT 0x6B
#define T2INT   0x73
```

An interrupt handler for the 80C552 timer 2 interrupt (T2INT) could be installed either by using the declarations above and writing:

```
ROM_VECTOR(T2INT, t2int_handler);
```

or by directly using the vector address:

```
ROM_VECTOR(0x73, t2int_handler);
```

Table 11.8: Interrupt vector names

Name	Vector	Interrupting device
`EXTI0`	034	External Interrupt 0
`TIMER0`	0BH	Timer 0
`EXTI1`	13H	External Interrupt 1
`TIMER1`	1BH	Timer 1
`SINT`	23H	Onboard serial port
`TIMER2`	2BH	Timer 2 (8052 only)

11.11 Mixing C and 8051 assembler code

8051 assembly language code can be mixed with C code using three different techniques.

11.11.1 External Assembly Language Functions

Entire functions may be coded in assembly language, assembled by AS51 as separate *.as* source files and combined into the binary image using the linker. This technique allows arguments and return values to be passed between C and assembler code.

To access an external function, first include an appropriate C *extern* declaration in the calling C code. For example, suppose you need an assembly language function to provide access to the rotate left instruction on the 8051:

```
extern char rotate_left(char);
```

declares an external function called *rotate_left()* which has a return value type of *char* and takes a single argument of type *char*. The actual code for *rotate_left()* will be supplied by an external *.as* file which will be separately assembled with AS51.

The full 8051 assembler code for *rotate_left()* would be something like:

```
PSECT   text,class=CODE
GLOBAL  _rotate_left
SIGNAT  _rotate_left,4201
PSECT   text
_rotate_left:
MOV     A,R5
RL      A
MOV     R3,A
```

```
RET
```

The name of the assembly language function is the name declared in C, with an underscore prepended. The *GLOBAL* directive is the assembler equivalent to the C *extern* keyword and the *SIGNAT* directive is used to enforce link time calling convention checking. Signature checking and the *SIGNAT* directive are discussed in more detail later in this chapter.

Note that in order for assembly language functions to work properly they must look in the right place for any arguments passed and must correctly set up any return values. In the example above, the R5 register was used for the argument to the function, and the R3 register was used for the return value. In small and medium model, the compiler uses a combination of register based argument passing and static allocation of arguments and local variables. Local variable allocation, argument and return value passing mechanisms are discussed in detail later in the manual. They should be understood before attempting to write assembly language routines.

11.11.2 Accessing C objects from within assembler

Global C objects may be directly accessed from within assembly code using their name prepended with an *underscore* character. For example, the object `foo` defined globally in a C module:

```
near char foo;
```

may be access from assembler as follows.

```
GLOBAL  _foo
mov r0,_foo
```

If the assembler is contained in a different module, then the GLOBAL assembler directive should be used in the assembler code to make the symbol name available, as above. If the object is being accessed from in-line assembly in another module, then an `extern` declaration for the object can be made in the C code, for example:

```
extern near char foo;
```

This declaration will only take effect in the module if the object is also accessed from within C code. If this is not the case then, an in-line GLOBAL assembler directive should be used.

11.11.3 #asm, #endasm and asm()

8051 instructions may also be directly embedded in C code using the directives *#asm*, *#endasm* and *asm()*. The *#asm* and *#endasm* directives are used to start and end a block of assembler instructions

which are to be embedded inside C code. The *asm()* directive is used to embed a single assembler instruction in the code generated by the C compiler.

To continue our example from above, you could directly code a rotate left on a memory byte using either technique, as the following example shows:

```
#include <stdio.h>
unsigned char  var;
main()
{
        var = 1;
        printf("var = 0x%2.2X\n", var);
#asm
        MOV     A,_var
        RL      A
        MOV     _var,A
#endasm
        printf("var = 0x%2.2X\n", var);
        asm("MOV  A,_var");
        asm("RL A");
        asm("MOV _var,A");
        printf("var = 0x%2.2X\n", var);
}
```

When using inline assembler code, great care must be taken to avoid interacting with compiler generated code. If in doubt, compile your program with the C51 **-S** option and examine the assembler code generated by the compiler.

IMPORTANT NOTE: the *#asm* and *#endasm* construct is not syntactically part of the C program, and thus it does NOT obey normal C flow-of-control rules. For example, you cannot use a *#asm* block with an if statement and expect it to work correctly. If you use in-line assembler around any C constructs such as if, while, do etc. they you should use only the *asm("")* form, which is a C statement and will correctly interact with all C flow-of-control structures.

11.12 Preprocessing

All C source files are preprocessed before compilation. Assembler files can also be preprocessed if the `-p` command-line option is issued.

11.12.1 Preprocessor Directives

C51 accepts several specialised preprocessor directives in addition to the standard directives. These are listed in Table 11.9.

Table 11.9 Preprocessor directives

Directive	Meaning	Example
#	preprocessor null directive, do nothing	#
#assert	generate error if condition false	#assert SIZE > 10
#asm	signifies the beginning of in-line assembly	#asm inc dptr #endasm
#define	define preprocessor macro	#define SIZE 5 #define FLAG #define add(a,b) ((a)+(b))
#elif	short for #else #if	see #ifdef
#else	conditionally include source lines	see #if
#endasm	terminate in-line assembly	see #asm
#endif	terminate conditional source inclusion	see #if
#error	generate an error message	#error Size too big
#if	include source lines if constant expression true	#if SIZE < 10 c = process(10) #else skip(); #endif
#ifdef	include source lines if preprocessor symbol defined	#ifdef FLAG do_loop(); #elif SIZE == 5 skip_loop(); #endif
#ifndef	include source lines if preprocessor symbol not defined	#ifndef FLAG jump(); #endif
#include	include text file into source	#include <stdio.h> #include "project.h"
#line	specify line number and filename for listing	#line 3 final
#nn	(where *nn* is a number) short for #line *nn*	#20
continued…		

Directive	Meaning	Example
#pragma	compiler specific options	See Section 11.12.3
#undef	undefines preprocessor symbol	#undef FLAG
#warning	generate a warning message	#warning Length not set

Macro expansion using arguments can use the # character to convert an argument to a string, and the ## sequence to concatenate tokens.

11.12.2 Predefined Macros

The compiler drivers define certain symbols to the preprocessor (CPP), allowing conditional compilation based on chip type and other parameters. The symbols listed in Table 11.10 show the more common symbols defined by the drivers. Each symbol, if defined, is equated to 1 unless otherwise stated.

11.12.3 Pragma Directives

There are certain compile-time directives that can be used to modify the behaviour of the compiler. These are implemented through the use of the ANSI standard #pragma facility. The format of a pragma is:

```
#pragma keyword options
```

where ***keyword*** is one of a set of keywords, some of which are followed by certain ***options***. A list of the keywords is given in Table 11.11. Those keywords not discussed elsewhere are detailed below.

11.12.3.1 The #pragma jis and nojis Directives

If your code includes strings with two-byte characters in the JIS encoding for Japanese and other national characters, the #pragma jis directive will enable proper handling of these characters, specifically not interpreting a *backslash* “\” character when it appears as the second half of a two byte character. The nojis directive disables this special handling. JIS character handling is disabled by default.

11.12.3.2 The #pragma printf_check Directive

Certain library functions accept a format string followed by a variable number of arguments in the manner of printf(). Although the format string is interpreted at run-time, it can be compile-time checked for consistency with the remaining arguments. This directive enables this checking for the named function, e.g. the system header file `<stdio.h>` includes the directive #pragma

Table 11.10: Predefined CPP symbols

Symbol	When set	Usage
HI_TECH_C	Always	To indicate that the compiler in use is HI-TECH C.
_HTC_VER_MAJOR_	Always	To indicate the integer component of the compiler's version number.
_HTC_VER_MINOR_	Always	To indicate the decimal component of the compiler's version number.
_HTC_VER_PATCH_	Always	To indicate the patch level of the compiler's version number.
LARGE_DATA	-Bm, -Bl, -Bh	To indicate that extern and static variables are by default allocated in external RAM.
SMALL_DATA	-Bs	To indicate that extern and static variables are by default allocated in internal RAM.
HUGE_MODEL	-Bh	To indicate that code is compiled in huge memory model.
LARGE_MODEL	-Bl	To indicate that code is compiled in large memory model.
MEDIUM_MODEL	-Bm	To indicate that code is compiled in medium memory model.
SMALL_MODEL	-Bs	To indicate that code is compiled in small memory model.
i8051	Always	To indicate that this is an 8051 device.
_XXXXX	When chip selected	To indicate the specific chip type selected.
__FILE__	Always	To indicate this source file being preprocessed.
__LINE__	Always	To indicate this source line number.
__DATE__	Always	To indicate the current date, e.g. May 21 2004
__TIME__	Always	To indicate the current time, e.g. 08:06:31.

printf_check(printf) const to enable this checking for `printf()`. You may also use this for any user-defined function that accepts printf-style format strings. The qualifier following the function name is to allow automatic conversion of pointers in variable argument lists. The above example would cast any pointers to strings in RAM to be pointers of the type (`const char *`)

Note that the warning level must be set to -1 or below for this option to have effect.

11.12.3.3 The #pragma psect Directive

Normally the object code generated by the compiler is broken into the standard psects as already documented. This is fine for most applications, but sometimes it is necessary to redirect variables or code into different psects when a special memory configuration is desired. For example, if the hardware includes an area of memory which is battery backed, it may be desirable to redirect certain variables from *bss* into a psect which is not cleared at startup. Code and data for any of the standard C psects may be redirected using a `#pragma psect` directive. For example, if all executable code generated by a particular C source file is to be placed into a psect called *altcode*,the following directive should be used:

```
#pragma psect  text=altcode
```

This directive tells the compiler that anything which would normally be placed in the *text* psect should now be placed in the *altcode* psect. Any given psect should only be redirected once in a particular source file, and all psect redirections for a particular source file should be placed at the top of the file, below any #includes and above any other declarations.

For example, to declare a group of uninitialized variables which are all placed in a psect called *nvram*, the following technique should be used:

```
---File NVRAM.C

#pragma psect   bss=nvram
char    buffer[20];
int     var1, var2, var3;
```

Any files which need to access the variables defined in NVRAM.C should `#include` the following header file:

```
--File NVRAM.H

extern char     buffer[20];
extern int      var1, var2, var3;
```

Table 11.11: Pragma directives

Directive	Meaning	Example
`interrupt_level`	Allow interrupt function to be called from main-line code. See Section 11.10.2	`#pragma interrupt_level 1`
`jis`	Enable JIS character handling in strings	`#pragma jis`
`nojis`	Disable JIS character handling (default)	`#pragma nojis`
`printf_check`	Enable printf-style format string checking	`#pragma printf_check(printf) const`
`psect`	Rename compiler-defined psect	`#pragma psect text=mytext`
`regsused`	Specify registers which are used in an interrupt	`#pragma regsused r0`
`strings`	Define constant string qualifiers	`#pragma strings code`
`switch`	Specify code generation for switch statements	`#pragma switch direct`

The `#pragma psect` directive allows code and data to be split into arbitrary memory areas. Definitions of code or data for non-standard psects should be kept in separate source files as documented above. When you link code which uses non-standard psect names, you will not be able to use the C51 -A option to specify the link addresses for the new psects. Instead, you will need to use the C51 -L option to specify an extra linker option.

If you want a nearly standard configuration with the addition of only an extra psect like *nvram*, you can use the C51 -L option to add an extra -P specification to the linker command.

For example:

```
c51 --chip=8051 -Bm -L-Pnvram=1000h/20000h --ram=2000-2fff test.obj nv.obj
```

will link *test.obj* and *nv.obj* with a standard configuration of ROM at 0h, internal RAM at 20h, external RAM at 2000h and the extra *nvram* psect at 1000h in RAM, but not overlapping any valid ROM load address.

Table 11.12: Valid regsused register names

Register Name	Description
`r0..r7`	bank 0 general purpose registers
`8..15`	address of bank 1 general purpose registers
`a`	accumulator
`b`	B register
`dph, dpl, dptr`	data pointer: high, low, both

11.12.3.4 The #pragma regsused Directive

C51 will automatically save context when an interrupt occurs. The compiler will determine only those registers and objects which need to be saved for the particular interrupt function defined. The `#pragma` regsused directive allows the programmer to further limit the registers and objects that the compiler might save and retrieve on interrupt.

Table 11.12 shows registers names that would commonly be used with this directive. The register names are not case sensitive and a warning will be produced if the register name is not recognised. Note that the names in bank 1 represent the memory address of the register.

This pragma affects the first interrupt function following in the source code. Code which contains multiple interrupt functions should include one directive for each interrupt function.

For example, to limit the compiler to saving no registers for an interrupt function other than the accumulator, B register, and the R0 and R1 registers in banks 0 and 1, use:

```
#pragma regsused a,b,r0,r1,8,9
```

Even if a register other than these has been used and that register would normally be saved, it will *not* be saved if this pragma is in effect. The registers will only be automatically saved by the compiler if required.

11.12.3.5 The #pragma strings Directive

Any user-defined variables can be qualified by a number of type qualifiers (see Sections 11.3.8 and 11.3.9) but constant strings (i.e. anonymous strings embedded in expressions) are normally unqualified. This means they will be put into the data segment. To control this behaviour, the `#pragma psect` strings directive allows you to specify a set of qualifiers to be applied to all subsequent constant strings. If a qualifier is specified, it will be added to any qualifiers specified previously. Using the directive without a qualifier will remove all qualifiers from any subsequent strings, i.e. restore to normal.

For example., to qualify strings with `code` you should use the example given in Table 11.11. Note that all constant strings will then have type `code char *` and will not be usable where a simple `char *` type is expected.

11.12.3.6 The #pragma switch Directive

Normally the compiler decides the code generation method for switch statements which results in the smallest possible code size. Specifying the `direct` option to the `#pragma switch` directive forces the compiler to generate the table look-up style switch method. This is mostly useful where either timing or code size is an issue for switch statements (ie: state machines) and a jump table is preferred over direct comparison or vice versa. This pragma affects all code generated onwards. The `auto` option may be used to revert to the default behaviour.

11.13 Linking programs

The compiler will automatically invoke the linker unless requested to stop after producing assembler code (C51 -S option) or object code (C51 -C option).

C51 by default generates *Intel hex* files. If you use the –OUTPUT=bin option or specify an output file with a *.bin* file type using the C51 -O option, the compiler will generate a binary image instead. The file will contain code starting from the lowest initialized address in the program. For example:

```
c51 --chip=8051 -v -oxx.bin
```

will produce a binary file starting with the RESET vector at address 0H, followed by the other interrupt vectors, user code, initialized data and library code.

When producing code which is to be downloaded using a debugger, the *codeoffset* value specified should be the address of the area in RAM where the downloaded code will be located. Code which is to be run using a simulator should be compiled using the normal addresses for one of the 8051 variants which the simulator supports.

After linking, the compiler will automatically generate a memory usage map which shows the address and size of all memory areas which are used by the compiled code. For example:

```
Memory Usage Map:
Program space:
    CODE          used    B3h (   179) of 10000h bytes (  0.3%)
Internal Data:
    BITSEG        used     0h (     0) of    80h bits  (  0.8%)
    DATA          used    23h (    35) of    E0h bytes ( 15.6%)
```

```
External Data:
    XDATA         used     2h (     2) of  FF00h bytes (  0.0%)
Summary:
    Program space used    B3h (   179) of 10000h bytes (  0.3%)
    Internal Data used    24h (    36) of   100h bytes ( 14.1%)
    External Data used     2h (     2) of 10000h bytes (  0.0%)
```

More detailed memory usage information, listed in ascending order of individual psects, may be obtained by using the C51 –summary=all option.

11.13.1 Replacing Library Modules

Although C51 comes with a librarian (`LIBR`) which allows you to unpack a library files and replace modules with your own modified versions, you can easily replace a module within a library without having to do this. If you add the source file which contains the library routine you wish to replace on the command-line list of source files then the routine will replace the routine in the library file with the same name. For example, if you wished to make changes to the library function `max()` which resides in the file `max.c` in the SOURCES directory, you could make a copy of this source file, make the appropriate changes and then compile and use it as follows.

```
c51 --chip=8051 main.c init.c max.c
```

The code for `max()` in `max.c` will be linked into the program rather than the `max()` function contained in the standard libraries. Note, that if you replace an assembler module, you may need the `-P` option to preprocess assembler files as the library assembler files often contain C preprocessor directives.

11.13.2 Signature checking

The compiler automatically produces signatures for all functions. A signature is a 16 bit value computed from a combination of the function's return data type, the number of its parameters and other information affecting the calling sequence for the function. This signature is output in the object code of any function referencing or defining the function.

At link time the linker will report any mismatch of signatures. Thus if a function is declared in one module in a different way (for example, as *char* instead of *short*) then the linker will report an error.

It is sometimes necessary to write assembly language routines which are called from C using an *extern* declaration. Such assembly language functions need to include a signature which is compatible with the C prototype used to call them. The simplest method of determining the correct signature

for a function is to write a dummy C function with the same prototype and compile it to assembly language using the C51 **-S** option.

For example, suppose you have an assembly language routine called *_widget* which takes two *int* arguments and returns a *char* value. The prototype used to call this function from C would be:

```
extern char widget(int, int);
```

Where a call to _widget is made in the C code, the signature for a function with two *int* arguments and a *char* return value would be generated. In order to match the correct signature the source code for widget needs to contain an AS51 *SIGNAT* directive which defines the same signature value. To determine the correct value, you would write the following code:

```
char widget(int arg1, int arg2)
{
}
```

Now compile it to assembler code using:

```
c51 --chip=8051 -S x.c
```

The resultant assembler code includes the following line:

```
signat  _widget,8249
```

The *SIGNAT* directive tells the assembler to include a record in the *.obj* file which associates the value 8249 with symbol *_widget***.** The value 8249 is the correct signature for a function with two *int* arguments and a *char* return value. If this line is copied into the *.as* file where *_widget* is defined, it will associate the correct signature with the function and the linker will be able to check for correct argument passing.

For example, if another *.c* file contains the declaration:

```
extern char widget(long);
```

a different signature will be generated and the linker will report a signature mismatch. This will alert you to the possible existence of incompatible calling conventions.

11.13.3 Linker-Defined Symbols

The link address of a psect can be obtained from the value of a global symbol with name ***__****Lname* where *name* is the name of the psect. For example, ***__****Lbss* is the low bound of the *bss* psect. The highest address of a psect (i.e. the link address plus the size) is symbol ***__****Hname*. If the psect has different load and link addresses, as may be the case if the *data* psect is linked for RAM operation, the load address is ***__****Bname*.

Table 11.13: Console I/O functions

Function	Purpose
`void init_uart(void);`	Initialise the console (Serial port)
`void putch(char ch);`	Write character to the console
`char getch(void);`	Get a character from the console
`char getche(void);`	Get and echo a console character
`int kbhit(void);`	Returns 1 if a character is available

11.14 Standard I/O Functions and Serial I/O

In order to use the standard I/O functions (*printf()*, *puts()*, *scanf()*, gets(), etc.), you will need to implement library routines which implement low level console I/O on your target hardware. This is usually achieved by communicating via a serial port. All standard I/O routines perform character I/O by calling the *<conio.h>* routines listed in Table 11.13.

The generic *51xxNxC.lib* libraries are supplied with standard versions of these routines installed. If you attempt to run a program which uses console I/O before you have customised an appropriate console I/O module in the library, it will probably not work. The SOURCES directory includes a source file *getch.c* which implements console I/O via standard 8051 serial port. These routines will probably require modification to the baud rate initialization in *init_uart()*. See Section 11.13.1 for information on easily replacing library modules.

If you are using a HI-TECH Software debugger or simulator, use the console I/O routines supplied with the debugger when creating code which is to be downloaded.

11.15 Optimizing Code for the 8051

Due to the limitations imposed by the 8051 instruction set, care needs to be taken to avoid writing code which will be large or inefficient. To improve execution speed and reduce code size, some or all of these suggestions can be used:

- Use *char*, *signed char* or *unsigned char* types instead of *int* wherever possible. The 8051 can manipulate 8 bit quantities much more efficiently than 16 bit quantities.

- In the small and medium memory models, local *auto* variables always reside in internal RAM and can be manipulated using the direct addressing mode. Since storage used by function arguments and *auto* variables can be reused by other functions, try to use *auto* variables where

possible to reduce internal RAM usage. Any variable which is declared within a function and which is not *static* is an *auto* variable and will be placed in internal RAM.

- In the medium, large, and huge models, variables which are critical to performance should be declared *near*, which places them in internal RAM.
- Variables which are less critical to performance, but still frequently accessed, can be placed in the indirectly accessible area from 80H to FFH using the *idata* qualifier. *Idata* variables are slower to access than normal internal RAM variables, but are more efficient than variables in external RAM.
- Choose a memory model which is applicable to the application. If you have no need for recursive or re-entrant code then you should be using either small or medium model.
- Use small model for applications which require less than around 200 bytes of variables and no large buffers. Small model applications which need only a small amount of external RAM for buffers may be written using the *far* qualifier to declare external RAM variables.
- Medium model should be used for applications which require a large number of static variables and buffers. Performance critical variables should be placed in internal RAM by making them local, or using the *near* and *idata* qualifiers.
- Pointer manipulation can be improved substantially by using pointers of class *pointer to near* or *pointer to idata* wherever possible. *Near* and *idata* pointers occupy only a single byte of storage and can only address objects in internal RAM. Near pointers can be easily dereferenced using the register indirect mode of the 8051, while normal pointers frequently require time consuming library calls.
- You can declare a *pointer to near* by including the *near* qualifier anywhere to the left of the "*" in the declaration, for example: `near char * nptr;` declares a pointer to a *near* char. Similarly a *pointer to idata* may be declared as: `idata char * iptr;`
- If you require a pointer which you know will only ever address objects in internal RAM use declarations such as the ones above to maximise the efficiency of your code.
- Use unsigned types like *unsigned char* wherever possible as the 8051 handles unsigned quantities more readily than signed quantities. Remember that the default behaviour of *char* is signed.

Each of the techniques listed above should gain you some ROM space and improve execution speed. If all of these techniques are used to their fullest, the compiler will produce very good code indeed. The 8051 imposes some limitations but if used intelligently the HI-TECH compiler will give results which frequently could not be improved with hand coding of assembly language.

Chapter 12

Macro Assembler

The HI-TECH Software 8051 Macro Assembler assembles source files for the Intel 8051 family of microprocessors.

This chapter describes the usage of the assembler and the directives (assembler pseudo-ops) accepted by the assembler. The 8051 instruction set, listing all mnemonics, opcodes and addressing forms, is listed at the end of this chapter.

For a description of the available special function registers and any extra instructions refer to the appropriate processor handbook.

The HI-TECH assembler package includes a linker, librarian, cross reference generator and an object code converter.

12.1 Assembler Usage

The assembler is called AS51 and is available to run on PC and UNIX operating systems.

The usage of the assembler is similar under all of these operating systems. All command line options are recognised in either upper or lower case. The basic command format is shown is:

```
as51 [ options ] files ...
```

Files is a space-separated list of one or more assembler source files. Where more than one source file is specified the assembler treats them as a single module, i.e. a single assembly will be performed on the concatenation of all the source files specified. The files must be specified in full, no default extensions or suffixes are assumed.

Options is an optional space separated list of assembler options, each with a minus sign (-) as the first character. A full list of possible options is given in Table 12.1, and a full description of each option follows.

Table 12.1: AS51 command-line options

Option	Meaning	Default
`-A`	80C751 code (`AJMP/ACALL`)	8051 CODE
`-Q`	Quick assembly	Optimized assembly
`-U`	No undef'd symbol messages	
`-S`	No size error messages	
`-X`	No local symbols in OBJ file	
`-O`*`outfile`*	Specify object name	srcfile.OBJ
`-L`*`listfile`*	Produce listing	No listing
`-W`*`width`*	Specify listing page width	80 or 132
`-F`*`length`*	Specify listing form length	66
`-I`	List macro expansions	Don't list macros
`-C`	Produce cross-reference	No cross reference
`-V`	Include assembler line numbers in object file	No line numbers

12.2 Assembler options

The command line options recognised by AS51 are as follows:

-A The Philips/Signetics 80C751 series of processors do not support the `LJMP` and `LCALL` instructions. If the `-A` option is used, AS51 will assemble these instructions to `AJMP` and ACALL respectively. This assembler option is used by the C compiler when generating 80C751 code.

-Q The default mode of operation of the assembler is to iterate over the source code until the smallest possible code is produced, by optimizing jumps. If the `-Q` option is used then only two passes over the source code will be made, thereby speeding up assembly. This may result in `NOP` instructions being generated in the code where an optimization was performed on the second pass but not the first.

-U Undefined symbols encountered during assembly are treated as external, however an error message is issued for each undefined symbol unless the -U option is given. Use of this option suppresses the error messages only, it does not change the generated code.

-S If a byte-size memory location is intialized with a value which is too large to fit in 8 bits, then the assembler will generate a "Size error" message. Use of the `-S` option will suppress these messages.

-X The object file created by the assembler contains symbol information, including local symbols, i.e. symbols that are neither public or external. The `-X` option will prevent the local symbols from being included in the object file, thereby reducing the file size.

-O*outfile* By default the assembler determines the name of the object file to be created by stripping any suffix or extension (i.e. the portion after the last dot) from the first source file name and appending `.obj`. The `-O` option allows the user to override the default and specify and explicit filename for the object file.

-L*listfile* This option requests the generation of an assembly listing. If listfile is specified then the listing will be written to that file, otherwise it will be written to the standard output.

-W*width* This option allows specification of the listfile paper width, in characters. Width should be a decimal number greater than 41. The default width is 80 characters if the listfile is a device (terminal, printer etc.) or 132 if it is a file.

-F*length* The default listing pagelength is 66 lines (11 inches at 6 lines per inch). The -F option allows a different page length to be specified.

-I This option overrides any `NOLIST` assembler controls and forces listing of macro expansions and unassembled conditionals.

-C A cross reference file will be produced when this option is used. This file, called *`srcfile`*`.crf` where *`srcfile`* is the base portion of the first source file name, will contain raw cross reference information. The cross reference utility `CREF` must then be run to produce the formatted cross reference listing.

-V Include assembler line numbers and file names in the object file, for debugging purposes.

12.3 8051 Assembly language

The source language accepted by the HI-TECH Software 8051 Macro Assembler is described below. All opcode mnemonics and operand syntax are strictly as described in the Intel MCS-51 Programmer's Guide.

12.3.1 Character set

The character set used is standard 7 bit ASCII. Alphabetic case is significant for identifiers, but not opcodes and reserved words. Tabs are treated as equivalent to spaces.

Table 12.2: AS51 numbers and bases

Radix	Format
Binary	digits 0 and 1 followed by *B*
Octal	digits 0 to 7 followed by *O, Q, o* or *q*
Decimal	digits 0 to 9 followed by *D, d* or nothing
Hexadecimal	digits 0 to 9, A to F preceded by *Ox or followed by H or h*

12.3.2 Numbers

The assembler performs all arithmetic as signed 32 bit. Errors will be caused if a quantity is too large to fit in a memory location. The default radix for all numbers is 10. Other radices may be specified by a trailing base specifier as given in Table 12.2.

Hexadecimal numbers must have a leading digit to differentiate them from identifiers. Hexadecimal constants are accepted in either upper or lower case.

Note that a binary constant must have an upper case B following it, as a lower case b is used for temporary (numeric) label backward references.

Real numbers are accepted in the usual format for DF directives only. The exponent and mantissa of a real number must be decimal. Real numbers are stored in IEEE 32 bit format.

12.3.3 Delimiters

All numbers and identifiers must be delimited by white space, non alphanumeric characters or the end of a line.

12.3.4 Identifiers

Identifiers are user-defined symbols representing memory locations or numbers. A symbol may contain any number of characters drawn from the alphabetics, numerics and the special characters dollar ($), question mark (?) and underscore(_). The first character of an identifier may not be numeric. The case of alphabetics is significant, e.g. *Fred* is not the same symbol as *fred*.

12.3.4.1 Assembler generated identifiers

Where a LOCAL directive is used in a macro block, the assembler will generate a unique symbol to replace each specified identifier in each expansion of that macro. These unique symbols will have the form *??nnnn* where *nnnn* is a 4 digit number. The user should avoid defining symbols with the same form.

12.3.4.2 Location counter

The current location within the active program section is accessible via the symbol `$`.

12.3.4.3 Predefined Identifiers

Some identifies representing registers, bits within registers, and interrupt vector locations have been predefined. These predefined identifiers are case-insensitive, therefore you cannot redefine one in a different case.

12.3.5 Strings

A string is a sequence of characters not including carriage return or newline, enclosed within matching quotes. Either single (’) or double (") quotes may be used, but the opening and closing quotes must be the same. A string used as an operand to a DB directive may be any length, but a string used as operand to an instruction must not exceed 1 or 2 characters, depending on the size of the operand required.

12.3.6 Temporary labels

The assembler implements a system of temporary labels (as distinct from the local labels used in macros) which relieves the programmer from creating new labels within a block of code. A temporary label is defined as a numeric string, and may be referenced by the same numeric string with either an ‘f’ or ‘b’ suffix. When used with an ‘f’ suffix, the label reference is the first label with the same number found by looking *f*orward from the current location, and conversely a ‘b’ will cause the assembler to look *b*ackward for the label.

For example:

```
entry:
    mov r0,ploc
1:
    mov a,@r0
    jz 1f    ;end of string
    inc r0
    cjne a,r2,1b
    sjmp 2f  ;found it
1:
    clr a    ;return zero
    ret
2:
```

```
dec r0
mov a,r0 ;return pointer
ret
```

Note that even though there are two 1: labels, no ambiguity occurs, since each is referred to uniquely. The *cjne 1b* refers to a label further back in the source code, while *jz 1f* refers to a label further forward. In general, to avoid confusion, it is recommended that within a routine you do not duplicate numeric labels.

12.3.7 Expressions

Expressions are made up of numbers, symbols, strings and operators. The available operators are listed in Table 12.3, in order of precedence. The usual rules governing the syntax of expressions apply.

The operators above may all be freely combined in both constant and relocatable expressions. The HI-TECH linker permits relocation of complex expressions, so the results of expressions involving relocatable identifiers may not be resolved until link time.

12.3.8 Statement format

Legal statement formats are shown in table Table 12.4. The second form is only legal with certain directives, such as MACRO, SET and EQU. The *label* field is optional and if present should contain one identifier. The *name* field is mandatory and should also contain one identifier.

12.3.9 Addressing modes

The assembler recognises all standard 8051 addressing modes. All SFRs and bit addresses are accepted by the assembler. Consult an Intel handbook for full information.

12.3.10 Program sections

Program sections, or *psects*, are a way of grouping together parts of a program even though the source code may not be physically adjacent in the source file, or even where spread over several source files. A psect is identified by a name and has several attributes. The psect directive is used to define psects. It takes as arguments a name and an optional comma-separated list of flags. See the Section 12.3.11.5 for full information. The assembler associates no significance to the name of a psect.

Table 12.3: AS51 operators

Operator	**Purpose**	**Precedence**
`NUL`	Test for null argument	8
`^`	Exponentation	7
`,/, MOD`	multiply divide modulus	6
`SHR, SHL`	shift right, shift left	6
`ROR, ROL`	rotate right, rotate left	6
`+, -`	plus, minus (unary or binary)	5
`HIGHWORD`	high 16 bits of dword operand	5
`HIGH`	high byte of word expression	5
`LOW`	low byte of word expression	5
`SEG`	segment part of address	5
`EQ,NE,GT,GE,LT,LE`	Relational operators	4
`=,<>,>,>=,<,<=`	Relational operators	4
`NOT`	bitwise inversion	3
`AND`	bitwise conjunction	2
`OR`	bitwise disjunction	1
`XOR`	exclusive OR	1

Table 12.4: AS51 statement formats

label:	*opcode*	*operands*	*;comment*
name	*pseudo-op*	*operands*	*;comment*
;comment only			

12.3.11 Assembler directives

Assembler directives, or *pseudo-ops*, are used in a similar way to opcodes, but either, do not generate code, or generate non-executable code, i.e. data bytes.The directives are listed in table Table 12.5, and detailed below.

12.3.11.1 PUBLIC

The PUBLIC directive takes a comma separated list of symbols defined in the current module and which are to be accessible to other modules at link time. Example:

```
PUBLIC    lab1,lab2,lab3
```

12.3.11.2 EXTRN

This is the complement of PUBLIC; it declares symbols which may then be referenced even though they are defined in another module. Example:

```
EXTRN    lab1,lab2,lab3
```

12.3.11.3 GLOBAL

GLOBAL is a combination of PUBLIC and EXTRN; it declares a list of symbols which, if defined within the current module, are made public, otherwise are made external. Example:

```
GLOBAL   lab1,lab2,lab3
```

12.3.11.4 END

END is optional, but if present should be at the very end of the program. It will terminate the assembly. If an expression is supplied as an argument, that expression will be used to define the start address of the program. Whether this is of any use will depend on the linker. For example:

```
END  start_label
```

12.3.11.5 PSECT

The PSECT directive declares or resumes a program section. It takes as arguments a name and optionally a comma separated list of flags. The allowed flags are detailed below. Once a psect has been declared it may be resumed later by simply giving its name as an argument to another psect directive; the flags need not be repeated. The psect flags are listed in Table 12.6.

Table 12.5: AS51 directives

Directive	**Purpose**
PUBLIC	Make symbols accessible to other modules
EXTRN	Allow reference to other modules symbols
GLOBAL	Public or extrn as appropriate
END	End assembly
PSECT	Declare or resume program selection
ORG	Set location counter
EQU	Define symbol value
SET	Re-define symbol value
DB	Define constant byte(s)
DW	Define constant word(s)
DF	Define constant real(s)
DS	Reserve storage
FNADDR	Inform linker that a function may be indirectly called
FNARG	Inform linker that evaluation of arguments for one function requires calling another
FNBREAK	Break call graph links
FNCALL	Inform linker that one function calls another
FNCONF	Supply call graph configuration info to linker
FNINDIR	Inform linker that all functions with a particular signature may be indirectly called
FNROOT	Inform linker that a function is the "root" of a call graph
FNSIZE	Inform linker of argument and local variable sizes for a function
IF	Conditional assembly
ELSE	Alternate conditional assembly
ENDIF	End conditional assembly
MACRO	Macro definition
ENDM	End macro definition
LOCAL	Define local tabs
REPT	Repeat a block of code n times
IRP	Repeat a block of code with a list
IRPC	Repeat a block of code with a character list
EXITM	Terminate macro expansion
SIGNAT	Define function signature

Table 12.6: Psect flags

Flag	Meaning
ABS	Psect is absolute
BIT	Psect holds bit objects
GLOBAL	Psect is global (default)
LOCAL	Psect is not global
OVRLD	Psect will overlap same psect in other modules
PURE	Psect is to be read-only
RELOC	Start psect on specified boundary
SIZE	Maximum size of psect
SPACE	Represents area in which psect will reside

BIT The `BIT` flag defines the current psect as being bit addressable. Any storage allocated in a `BIT` psect will be in bits, not bytes. For example, `DS 4` in a `BIT` psect will reserve 4 bits of storage.

PURE The `PURE` flag instructs the linker that this psect will not be modified at run time and may therefore, for example, be placed in ROM. This flag is of limited usefulness since it depends on the linker and target system enforcing it.

ABS `ABS` defines the current psect as being absolute, i.e. it is to start at location 0. This does not mean that this module's contribution to the psect will start at 0, since other modules may contribute to the same psect.

OVRLD A psect defined as `OVRLD` will have the contribution from each module overlaid, rather than concatenated at run time. `OVRLD` in combination with `ABS` defines a truly absolute psect, i.e. a psect within which any symbols defined are absolute.

GLOBAL A psect defined as global will be combined with other global psects of the same name from other modules at link time. `GLOBAL` is the default.

LOCAL A psect defined as `LOCAL` will not be combined with other local psects at link time, even if there are others with the same name. A local psect may not have the same name as any global psect, even one in another module.

SIZE The `SIZE` flag allows a maximum size to be specified for the psect, e.g. `SIZE=100h`. This will be checked by the linker after psects have been combined from all modules.

RELOC The `RELOC` flag allows specification of a requirement for alignment of the psect on a particular boundary, e.g. `RELOC=100h` would specify that this psect must start on an address that is a multiple of 100h.

SPACE The `SPACE` flag is used to differentiate areas of memory which have overlapping addresses, but which are distinct. Psects which are positioned in ROM and RAM have a different `SPACE` value to indicate that ROM address zero, for example, is a different location to RAM address zero.

Some examples of the use of the PSECT directive follow:

```
PSECT fred
PSECT bill,size=100h,global
PSECT joh,abs,ovrld
```

12.3.11.6 ORG

ORG changes the value of the location counter within the current psect. This means that the addresses set with ORG are relative to the base of the psect, which is not determined until link time.

•

The `ORG` directive does *not* necessarily move the location counter to the absolute address you specify as the operand.

The argument to ORG must be either an absolute value, or a value referencing the current psect. In either case the current location counter is set to the value determined by the argument. For example:

```
ORG 100h
```

will move the location counter to the beginning of the current psect plus 100h. The actual location will not be known until link time. It is not possible to move the location counter backward.

In order to use the ORG directive to set the location counter to an absolute value, the directive must be used from within an absolute, overlaid psect. For example:

```
PSECT absdata,abs,ovrld
    ORG 50h
```

12.3.11.7 EQU and SET

This pseudo-op defines a symbol and equates its value to an expression. For example:

```
assembly EQU 123h
```

The identifier *assembly* will be given the value 123h. EQU is legal only when the symbol has not previously been defined.

SET is identical to EQU except that it may be used to re-define a symbol.

12.3.11.8 DB and DW

These directives initialize storage, as bytes or words respectively. The argument to each is a list of expressions, each of which will be assembled into one byte or word. DB may also take a multi-character string as an argument. Each character of the string will be assembled into one memory location.

An error will occur if the value of an expression is too big to fit into the memory location, e.g. if the value 1020 is given as an argument to DB. Examples:

```
lab: DB'X',1,2,3,4,"A string",0
DW 23*10,alabel,0,'a'
```

12.3.11.9 DF

DF initializes memory double words as real numbers. Each number will occupy 32 bits (4 bytes) and will be stored in IEEE 32 bit format, high byte first.

```
pi:  DF 3.14159
     DF 3.3,3e10,-23
```

12.3.11.10 DS

This directive reserves, but does not initialize, memory locations. The single argument is the number of bytes to be reserved. Examples:

```
alabel: DS23
xlabel: DS2+3
```

12.3.11.11 FNADDR

This directive tells the linker that a function has its address taken, and thus could be called indirectly through a function pointer. For example:

```
FNADDR  _func1
```

tells the linker that *func1()* has its address taken.

12.3.11.12 FNARG

The directive:

```
FNARG   fun1,fun2
```

tells the linker that evaluation of the arguments to function *fun1* involves a call to *fun2*, thus the memory argument memory allocated for the two functions should not overlap.

For example, the C function call *fred(var1, bill(), 2);* will generate the assembler directive:

```
FNARG_fred,_bill
```

thereby telling the linker that *bill()* is called while evaluating the arguments for a call to *fred()*.

12.3.11.13 FNBREAK

This directive is used to break links in the call graph information. The form of this directive is as follows:

```
FNBREAK fun1,fun2
```

and is automatically generated when the interrupt_level pragma is used. It states that the link to `fun1()` in the call graph rooted at `fun2()` should not be followed when checking for functions that appear in multiple call graphs. `Fun2()` is typically `intlevel0` or `intlevel1` in compiler-generated code when the interrupt level pragma is used.

12.3.11.14 FNCALL

This directive takes the form:

```
FNCALL  fun1,fun2
```

FNCALL is usually used in compiler generated code. It tells the linker that function *fun1* calls function *fun2*. This information is used by the linker when performing call graph analysis. If you write assembler code which calls a C function, use the FNCALL directive to ensure that your assembler function is taken into account.

For example, if you have an assembler routine called *_fred* which calls a C routine called *foo()*, in your assembler code you should write:

```
FNCALL  _fred,_foo
```

12.3.11.15 FNCONF

The FNCONF directive is used to supply the linker with configuration information for a *call graph*. FNCONF is written as follows:

```
FNCONF  psect,auto,args
```

where *psect* is the psect containing the call graph, auto is the prefix on all *auto* variable symbol names and *args* is the prefix on all function argument symbol names. This directive normally appears in only one place, the runtime startoff code used by C compiler generated code.

For most memory models, the run-time startoff routines (*rt51–nm.as, rt51-ns.as* and *rt51a-ns.as*) routines should include the directive:

```
FNCONF  rbss,?a,?
```

telling the linker that the call graph is in the *rbss* psect, auto variable blocks start with *?a* and function argument blocks start with *?*.

For large model, there is a stack, so call graphing is not necessary in *rt–nl.as*.

12.3.11.16 FNINDIR

This directive tells the linker that a function performs an indirect call to another function with a particular signature (see the SIGNAT directive). The linker must assume worst case that the function could call any other function which has the same signature and has had its address taken (see the FNADDR directive). For example, if a function called *fred()* performs an indirect call to a function with signature 8249, the compiler will produce the directive:

```
FNINDIR _fred,8249
```

12.3.11.17 FNSIZE

The FNSIZE directive informs the linker of the size of the local variable and argument area associated with a function. These values are used by the linker when building the call graph and assigning addresses to the variable and argument areas. This directive takes the form:

```
FNSIZE  func,local,args
```

The named function has a local variable area and argument area as specified, for example:

```
FNSIZE  _fred, 10, 5
```

means the function *fred()* has 10 bytes of local variables and 5 bytes of arguments.

The function name arguments to any of the call graph associated directives may be local or global. Local functions are, of course, defined in the current module, but must be used in the call graph construction in the same manner as global names.

12.3.11.18 FNROOT

This directive tells the assembler that a function is a *root function* and thus forms the root of a call graph. It could either be the C *main()* function or an interrupt function. For example, the C main module produce the directive:

```
FNROOT  _main
```

12.3.11.19 IF, ELSE and ENDIF

These directives implement conditional assembly. The argument to IF should be an absolute expression. If it is non-zero, then the code following it up to the next matching ELSE or ENDIF will be assembled. If the expression is zero then the code up to the next matching ELSE or ENDIF will be skipped. At an ELSE the sense of the conditional compilation will be inverted, while an ENDIF will terminate the conditional assembly block. Example:

```
IF some_symbol
MOV A,@R0
ELSE
MOVX A,@DPTR
ENDIF
```

In this example, if *some_symbol* is non-zero, the first MOV instruction will be assembled but not the second. Conversely if *some_symbol* is zero, the MOVX will be assembled but not the first MOV will not. Conditional assembly blocks may be nested.

12.3.11.20 MACRO and ENDM

These directives provide for the definition of macros. The MACRO directive should be preceded by the macro name and followed by a comma separated list of formal parameters. When the macro is used, the macro name should be used in the same manner as a machine opcode, followed by a list of arguments to be substituted for the formal parameters. For example:

```
xch macro reg1, reg2;exchange registers
mov a,r&reg1&    ;save reg1
mov r&reg1,reg2  ;reg2 ---> reg1
mov r&reg2,a     ;restore reg2
ENDM
```

defines a macro *xch*. The macro invocation `xch 3,4` would expand to:

```
mov a,r3
mov r3,4
mov r4,a
```

The *&* character may be used to delimit an argument used in the coding of the macro, thus permitting the concatenation of macro parameters with other text, but is removed in the actual macro expansion. The *&* character need not be used if commas or spaces are delimiting the argument, but should be used at both ends if no other delimiters are available.

The NUL operator may be used within a macro to test a macro argument. A comment may be suppressed within the expansion of a macro (thus saving space in the macro storage) by opening the comment with a double semicolon (;;).

12.3.11.21 LOCAL

The LOCAL directive allows unique labels to be defined for each expansion of a given macro. Any symbols listed after the LOCAL directive will have a unique assembler generated symbol substituted for them when the macro is expanded. For example:

```
copy MACRO src,dst,cnt
LOCAL loop
       mov r0,src
       mov r1,dst
       mov r2,#cnt
loop:  mov a,@r0
       mov @r1,a
       inc r0
```

```
        inc r1
        djnz r2,loop
ENDM
```

defines a macro *copy* which when invoked as:

```
copy #inbuf,#procbuf,32
```

expands to:

```
        mov  r0,#inbuf
        mov  r1,#procbuf
        mov  r2,#32
??0001: mov  a,@r0
        mov  @r1,a
        inc  r0
        inc  r1
        djnz r2,??0001
```

12.3.11.22 REPT

The REPT directive temporarily defines an unnamed macro then expands it a number of times as determined by its argument. For example:

```
mov r0,#zbuf
clr a
REPT3
mov @r0,a
inc r0
ENDM
```

expands to:

```
mov r0,#zbuf
clr a
mov @r0,a
inc r0
mov @r0,a
inc r0
mov @r0,a
inc r0
```

12.3.11.23 IRP and IRPC

The IRP and IRPC directives operate similarly to REPT. However, instead of repeating the block a fixed number of times, it is repeated once for each member of an argument list. In the case of IRP the list is a conventional macro argument list, in the case or IRPC it is each character in one argument. For each repetition the argument is substituted for one formal parameter.

For example:

```
IRP arg,lab1,lab2,#23
mov @r0,arg
inc r0
ENDM
```

expands to:

```
mov @r0,lab1
inc r0
mov @r0,lab2
inc r0
mov @r0,#23
inc r0
```

The IRPC directive is similar, except it substitutes one character at a time from a string of non-space characters. For example:

```
IRPC arg,ABC
LOCAL lab
cjne a,#'arg',lab
ljmp case_&arg
lab:
ENDM
```

expands to:

```
cjne a,#'A',??0000
ljmp case_A
??0000:
cjne a,#'B',??0001
ljmp case_B
??0001:
cjne a,#'C',??0002
ljmp case_C
??0002:
```

12.3.11.24 SIGNAT

This directive is used to associate a 16 bit signature value with a label. At link time the linker checks that all signatures defined for a particular label are the same and produces an error if they are not. The SIGNAT directive is used by the HI-TECH C compiler to enforce link time checking of function prototypes and calling conventions.

Use the SIGNAT directive if you want to write assembly language routines which are called from C. For example:

```
SIGNAT  _fred,8194
```

will associate the signature value 8192 with symbol *_fred*. If a different signature value for *_fred* is present in any object file, the linker will report an error.

12.3.12 Macro invocations

When invoking a macro, the argument list must be comma separated. If it is desired to include a comma (or other delimiter such as a space) in an argument then angle brackets (< and >) may be used to quote the argument. In addition the exclamation mark (!) may be used to quote a single character. The character immediately following the exclamation mark will be passed into the macro argument even if it is normally a comment indicator.

If an argument is preceded by a percent sign (%), that argument will be evaluated as an expression and passed as a decimal number, rather than as a string. This is useful if evaluation of the argument inside the macro body would yield a different result.

12.3.13 Assembler controls

Control lines may be included in the assembler source to control such things as listing format. Each control line starts with a dollar ($) character which is followed by a white-space separated list of control keywords. These keywords have no significance anywhere else in the program. Some keywords may have a parameter after them, which is always enclosed in parentheses. Most control keywords have a positive and a negative form. All have two letter abbreviations, the negative form is constructed by prefixing the keyword or the abbreviation with NO.

A list of keywords is given in Table 12.7, and each is described further below.

12.3.13.1 PAGELENGTH(*n*)

This control keyword specifies the length of the listing form. The default is 66 (11 inches at 6 lines per inch).

Table 12.7: `AS51` assembler controls

Control name	Abbreviation	Default
PAGELENGTH(n)	PL	PL(66)
PAGEWIDTH(n)	PW	PW(120)
XREF/NOXREF	XR/NOXR	NOXR
COND/NOCOND	CO/NOCO	CO
EJECT	EJ	
GEN/NOGEN	GE/NOGE	NOGE
INCLUDE(pathname)	IC	
LIST/NOLIST	LI/NOLI	LI
SAVE/RESORE	SA/RS	
TITLE(string)	TT	

12.3.13.2 PAGEWIDTH(*n*)

`PAGEWIDTH` allows the listing line width to be set.

12.3.13.3 XREF

`XREF` is equivalent to the command line option `-C`, it causes the assembler to produce a raw cross reference file. The utility `CREF` should be used to actually generate the formatted cross-reference listing.

12.3.13.4 COND

When `COND` is in effect, lines of code not assembled because of conditional assembly will be listed. If `NOCOND` is in effect only those lines actually assembled will appear in the listing.

12.3.13.5 EJECT

`EJECT` causes a new page to be started in the listing. A control-L (form feed) character will also cause a new page when encountered in the source.

12.3.13.6 GEN

When `GEN` is in effect the code generated by macro expansions will be listed. If `NOGEN` is in effect only the macro call will appear in the listing.

12.3.13.7 INCLUDE(pathname)

This control causes the file specified by *pathname* to be textually included at that point in the listing. The `INCLUDE` control must be the last control keyword on the line.

12.3.13.8 LIST

`LIST` and `NOLIST` turn listing on and off respectively

12.3.13.9 SAVE and RESTORE

`SAVE` pushes the current state of the `LIST`, `COND` and `GEN` flags onto a stack. `RESTORE` pops the top of the stack off into the flags. It may be used to selectively control listing inside macros.

12.3.13.10 TITLE(*string*)

This control keyword defines a title to appear at the top of every listing page. The *string* should be enclosed in single or double quotes.

Chapter 13

Linker and Utilities

13.1 Introduction

HI-TECH C incorporates a relocating assembler and linker to permit separate compilation of C source files. This means that a program may be divided into several source files, each of which may be kept to a manageable size for ease of editing and compilation, then each source file may be compiled separately and finally all the object files linked together into a single executable program.

This chapter describes the theory behind and the usage of the linker. Note however that in most instances it will not be necessary to use the linker directly, as the compiler drivers (HPD or command line) will automatically invoke the linker with all necessary arguments. Using the linker directly is not simple, and should be attempted only by those with a sound knowledge of the compiler and linking in general.

If it is absolutely necessary to use the linker directly, the best way to start is to copy the linker arguments constructed by the compiler driver, and modify them as appropriate. This will ensure that the necessary startup module and arguments are present.

Note also that the linker supplied with HI-TECH C is generic to a wide variety of compilers for several different processors. Not all features described in this chapter are applicable to all compilers.

13.2 Relocation and Psects

The fundamental task of the linker is to combine several relocatable object files into one. The object files are said to be *relocatable* since the files have sufficient information in them so that any references to program or data addresses (e.g. the address of a function) within the file may be adjusted according to where the file is ultimately located in memory after the linkage process. Thus

the file is said to be relocatable. Relocation may take two basic forms; relocation by name, i.e. relocation by the ultimate value of a global symbol, or relocation by psect, i.e. relocation by the base address of a particular section of code, for example the section of code containing the actual executable instructions.

13.3 Program Sections

Any object file may contain bytes to be stored in memory in one or more program sections, which will be referred to as *psects*. These psects represent logical groupings of certain types of code bytes in the program. In general the compiler will produce code in three basic types of psects, although there will be several different types of each. The three basic kinds are text psects, containing executable code, data psects, containing initialised data, and bss psects, containing uninitialised but reserved data.

The difference between the data and bss psects may be illustrated by considering two external variables; one is initialised to the value 1, and the other is not initialised. The first will be placed into the data psect, and the second in the bss psect. The bss psect is always cleared to zeros on startup of the program, thus the second variable will be initialised at run time to zero. The first will however occupy space in the program file, and will maintain its initialised value of 1 at startup. It is quite possible to modify the value of a variable in the data psect during execution, however it is better practice not to do so, since this leads to more consistent use of variables, and allows for restartable and ROMable programs.

For more information on the particular psects used in a specific compiler, refer to the appropriate machine-specific chapter.

13.4 Local Psects

Most psects are `global`, i.e. they are referred to by the same name in all modules, and any reference in any module to a `global` psect will refer to the same psect as any other reference. Some psects are `local`, which means that they are local to only one module, and will be considered as separate from any other psect even of the same name in another module. `Local` psects can only be referred to at link time by a class name, which is a name associated with one or more psects via the `PSECT` directive `class=` in assembler code. See Section 12.3.11.5 for more information on `PSECT` options.

13.5 Global Symbols

The linker handles only symbols which have been declared as `GLOBAL` to the assembler. The code generator generates these assembler directives whenever it encounters global C objects. At the C

source level, this means all names which have storage class external and which are not declared as `static`. These symbols may be referred to by modules other than the one in which they are defined. It is the linker's job to match up the definition of a global symbol with the references to it. Other symbols (local symbols) are passed through the linker to the symbol file, but are not otherwise processed by the linker.

13.6 Link and load addresses

The linker deals with two kinds of addresses; *link* and *load* addresses. Generally speaking the link address of a psect is the address by which it will be accessed at run time. The load address, which may or may not be the same as the link address, is the address at which the psect will start within the output file (HEX or binary file etc.). In the case of the 8086 processor, the link address roughly corresponds to the offset within a segment, while the load address corresponds to the physical address of a segment. The segment address is the load address divided by 16.

Other examples of link and load addresses being different are; an initialised data psect that is copied from ROM to RAM at startup, so that it may be modified at run time; a banked text psect that is mapped from a physical (== load) address to a virtual (== link) address at run time.

The exact manner in which link and load addresses are used depends very much on the particular compiler and memory model being used.

13.7 Operation

A command to the linker takes the following form:

```
hlink[1] options files ...
```

Options is zero or more linker options, each of which modifies the behaviour of the linker in some way. *Files* is one or more object files, and zero or more library names. The options recognised by the linker are listed in Table 13.1 and discussed in the following paragraphs.

Table 13.1: Linker command-line options

Option	Effect
`-8`	Use 8086 style segment:offset address form
`-Aclass=low-high,...`	Specify address ranges for a class
continued...	

[1] In earlier versions of HI-TECH C the linker was called `LINK.EXE`

Table 13.1: Linker command-line options

Option	Effect
-C*x*	Call graph options
-C*psect=class*	Specify a class name for a global psect
-C*baseaddr*	Produce binary output file based at *baseaddr*
-D*class=delta*	Specify a class delta value
-D*symfile*	Produce old-style symbol file
-E*errfile*	Write error messages to *errfile*
-F	Produce .obj file with only symbol records
-G*spec*	Specify calculation for segment selectors
-H*symfile*	Generate symbol file
-H+*symfile*	Generate enhanced symbol file
-I	Ignore undefined symbols
-J*num*	Set maximum number of errors before aborting
-K	Prevent overlaying function parameter and auto areas
-L	Preserve relocation items in .obj file
-LM	Preserve segment relocation items in .obj file
-N	Sort symbol table in map file by address order
-Nc	Sort symbol table in map file by class address order
-Ns	Sort symbol table in map file by space address order
-M*mapfile*	Generate a link map in the named file
-O*outfile*	Specify name of output file
-P*spec*	Specify psect addresses and ordering
-Q*processor*	Specify the processor type (for cosmetic reasons only)
-S	Inhibit listing of symbols in symbol file
-S*class=limit[,bound]*	Specify address limit, and start boundary for a class of psects
-U*symbol*	Pre-enter symbol in table as undefined
-V*avmap*	Use file *avmap* to generate an *Avocet* format symbol file
-W*warnlev*	Set warning level (-9 to 9)
-W*width*	Set map file width (>=10)
-X	Remove any local symbols from the symbol file
-Z	Remove trivial local symbols from the symbol file

13.7.1 Numbers in linker options

Several linker options require memory addresses or sizes to be specified. The syntax for all these is similar. By default, the number will be interpreted as a decimal value. To force interpretation as a

hex number, a trailing H should be added, e.g. 765FH will be treated as a hex number.

13.7.2 -A*class=low-high,...*

Normally psects are linked according to the information given to a -P option (see below) but sometimes it is desired to have a class of psects linked into more than one non-contiguous address range. This option allows a number of address ranges to be specified for a class. For example:

```
-ACODE=1020h-7FFEh,8000h-BFFEh
```

specifies that the class CODE is to be linked into the given address ranges. Note that a contribution to a psect from one module cannot be split, but the linker will attempt to pack each block from each module into the address ranges, starting with the first specified.

Where there are a number of identical, contiguous address ranges, they may be specified with a repeat count, e.g.

```
-ACODE=0-FFFFhx16
```

specifies that there are 16 contiguous ranges each 64k bytes in size, starting from zero. Even though the ranges are contiguous, no code will straddle a 64k boundary. The repeat count is specified as the character x or * after a range, followed by a count.

13.7.3 -C*x*

These options allow control over the call graph information which may be included in the map file produced by the linker. The -CN option removes the call graph information from the map file. The -CC option only include the critical paths of the call graph. A function call that is marked with a * in a full call graph is on a critical path and only these calls are included when the -CC option is used. A call graph is only produced for processors and memory models that use a compiled stack.

13.7.4 -C*psect=class*

This option will allow a psect to be associated with a specific class. Normally this is not required on the command line since classes are specified in object files.

13.7.5 -D*class=delta*

This option allows the *delta* value for psects that are members of the specified class to be defined. The delta value should be a number and represents the number of bytes per addressable unit of objects within the psects. Most psects do not need this option as they are defined with a *delta* value.

13.7.6 -D*symfile*

Use this option to produce an old-style symbol file. An old-style symbol file is an ASCII file, where each line has the link address of the symbol followed by the symbol name.

13.7.7 -E*errfile*

Error messages from the linker are written to standard error (file handle 2). Under DOS there is no convenient way to redirect this to a file (the compiler drivers will redirect standard error if standard output is redirected). This option will make the linker write all error messages to the specified file instead of the screen, which is the default standard error destination.

13.7.8 -F

Normally the linker will produce an object file that contains both program code and data bytes, and symbol information. Sometimes it is desired to produce a symbol-only object file that can be used again in a subsequent linker run to supply symbol values. The `-F` option will suppress data and code bytes from the output file, leaving only the symbol records.

This option can be used when producing more than one hex file for situations where the program is contained in different memory devices located at different addresses. The files for one device are compiled using this linker option to produce a symbol-only object file; this is then linked with the files for the other device. The process can then be repeated for the other files and device.

13.7.9 -G*spec*

When linking programs using segmented, or bank-switched psects, there are two ways the linker can assign segment addresses, or *selectors*, to each segment. A *segment* is defined as a contiguous group of psects where each psect in sequence has both its link and load address concatenated with the previous psect in the group. The segment address or selector for the segment is the value derived when a segment type relocation is processed by the linker.

By default the segment selector will be generated by dividing the base load address of the segment by the relocation quantum of the segment, which is based on the `reloc=` flag value given to psects at the assembler level. This is appropriate for 8086 real mode code, but not for protected mode or some bank-switched arrangements. In this instance the `-G` option is used to specify a method for calculating the segment selector. The argument to `-G` is a string similar to:

```
A/10h-4h
```

where *A* represents the load address of the segment and `/` represents division. This means "Take the load address of the psect, divide by 10 hex, then subtract 4". This form can be modified by substituting *N* for *A*, `*` for `/` (to represent multiplication), and adding rather than subtracting a constant.

The token `N` is replaced by the ordinal number of the segment, which is allocated by the linker. For example:

```
N*8+4
```

means "take the segment number, multiply by 8 then add 4". The result is the segment selector. This particular example would allocate segment selectors in the sequence 4, 12, 20, ... for the number of segments defined. This would be appropriate when compiling for 80286 protected mode, where these selectors would represent LDT entries.

13.7.10 -H*symfile*

This option will instruct the linker to generate a symbol file. The optional argument *symfile* specifies a file to receive the symbol file. The default file name is `l.sym`.

13.7.11 -H+*symfile*

This option will instruct the linker to generate an *enhanced* symbol file, which provides, in addition to the standard symbol file, class names associated with each symbol and a segments section which lists each class name and the range of memory it occupies. This format is recommended if the code is to be run in conjunction with a debugger. The optional argument *symfile* specifies a file to receive the symbol file. The default file name is `l.sym`.

13.7.12 -J*errcount*

The linker will stop processing object files after a certain number of errors (other than warnings). The default number is 10, but the `-J` option allows this to be altered.

13.7.13 -K

For compilers that use a compiled stack, the linker will try and overlay function auto and parameter areas in an attempt to reduce the total amount of RAM required. For debugging purposes, this feature can be disabled with this option.

13.7.14 -I

Usually failure to resolve a reference to an undefined symbol is a fatal error. Use of this option will cause undefined symbols to be treated as warnings instead.

13.7.15 -L

When the linker produces an output file it does not usually preserve any relocation information, since the file is now absolute. In some circumstances a further "relocation" of the program will be done at load time, e.g. when running a .exe file under DOS or a `.prg` file under TOS. This requires that some information about what addresses require relocation is preserved in the object (and subsequently the executable) file. The -L option will generate in the output file one null relocation record for each relocation record in the input.

13.7.16 -LM

Similar to the above option, this preserves relocation records in the output file, but only segment relocations. This is used particularly for generating `.exe` files to run under DOS.

13.7.17 -M*mapfile*

This option causes the linker to generate a link map in the named file, or on the standard output if the file name is omitted. The format of the map file is illustrated in Section 13.9.

13.7.18 -N, -Ns and-Nc

By default the symbol table in the link map will be sorted by name. The `-N` option will cause it to be sorted numerically, based on the value of the symbol. The `-Ns` and `-Nc` options work similarly except that the symbols are grouped by either their *space* value, or class.

13.7.19 -O*outfile*

This option allows specification of an output file name for the linker. The default output file name is `l.obj`. Use of this option will override the default.

13.7.20 -P*spec*

Psects are linked together and assigned addresses based on information supplied to the linker via `-P` options. The argument to the `-P` option consists basically of *comma*-separated sequences thus:

```
-Ppsect=lnkaddr+min/ldaddr+min,psect=lnkaddr/ldaddr, ...
```

There are several variations, but essentially each psect is listed with its desired link and load addresses, and a minimum value. All values may be omitted, in which case a default will apply, depending on previous values.

The minimum value, *min*, is preceded by a + sign, if present. It sets a minimum value for the link or load address. The address will be calculated as described below, but if it is less than the minimum then it will be set equal to the minimum.

The link and load addresses are either numbers as described above, or the names of other psects or classes, or special tokens. If the link address is a negative number, the psect is linked in reverse order with the top of the psect appearing at the specified address minus one. Psects following a negative address will be placed before the first psect in memory. If a link address is omitted, the psect's link address will be derived from the top of the previous psect, e.g.

```
-Ptext=100h,data,bss
```

In this example the text psect is linked at 100 hex (its load address defaults to the same). The `data` psect will be linked (and loaded) at an address which is 100 hex plus the length of the `text` psect, rounded up as necessary if the data psect has a `reloc=` value associated with it. Similarly, the `bss` psect will concatenate with the `data` psect. Again:

```
-Ptext=-100h,data,bss
```

will link in ascending order `bss`, data then `text` with the top of text appearing at address 0ffh.

If the load address is omitted entirely, it defaults to the same as the link address. If the *slash* `/` character is supplied, but no address is supplied after it, the load address will concatenate with the previous psect, e.g.

```
-Ptext=0,data=0/,bss
```

will cause both `text` and `data` to have a link address of zero, text will have a load address of 0, and data will have a load address starting after the end of `text`. The bss psect will concatenate with `data` for both link and load addresses.

The load address may be replaced with a *dot* `.` character. This tells the linker to set the load address of this psect to the same as its link address. The link or load address may also be the name of another (already linked) psect. This will explicitly concatenate the current psect with the previously specified psect, e.g.

```
-Ptext=0,data=8000h/,bss/. -Pnvram=bss,heap
```

This example shows `text` at zero, `data` linked at 8000h but loaded after `text`, `bss` is linked and loaded at 8000h plus the size of `data`, and `nvram` and `heap` are concatenated with `bss`. Note here the use of two `-P` options. Multiple `-P` options are processed in order.

If -A options have been used to specify address ranges for a class then this class name may be used in place of a link or load address, and space will be found in one of the address ranges. For example:

```
-ACODE=8000h-BFFEh,E000h-FFFEh
-Pdata=C000h/CODE
```

This will link `data` at C000h, but find space to load it in the address ranges associated with `CODE`. If no sufficiently large space is available, an error will result. Note that in this case the `data` psect will still be assembled into one contiguous block, whereas other psects in the class `CODE` will be distributed into the address ranges wherever they will fit. This means that if there are two or more psects in class `CODE`, they may be intermixed in the address ranges.

Any psects allocated by a `-P` option will have their load address range subtracted from any address ranges specified with the `-A` option. This allows a range to be specified with the `-A` option without knowing in advance how much of the lower part of the range, for example, will be required for other psects.

13.7.21 -Q*processor*

This option allows a processor type to be specified. This is purely for information placed in the map file. The argument to this option is a string describing the processor.

13.7.22 -S

This option prevents symbol information relating from being included in the symbol file produced by the linker. Segment information is still included.

13.7.23 -S*class=limit[, bound]*

A class of psects may have an upper address *limit* associated with it. The following example places a limit on the maximum address of the `CODE` class of psects to one less than 400h.

```
-SCODE=400h
```

Note that to set an upper limit to a psect, this must be set in assembler code (with a `limit=` flag on a `PSECT` directive).

If the *bound* (boundary) argument is used, the class of psects will start on a multiple of the bound address. This example places the `FARCODE` class of psects at a multiple of 1000h, but with an upper address limit of 6000h:

```
-SFARCODE=6000h,1000h
```

13.7.24 -U*symbol*

This option will enter the specified symbol into the linker's symbol table as an undefined symbol. This is useful for linking entirely from libraries, or for linking a module from a library where the ordering has been arranged so that by default a later module will be linked.

13.7.25 -V*avmap*

To produce an *Avocet* format symbol file, the linker needs to be given a map file to allow it to map psect names to *Avocet* memory identifiers. The avmap file will normally be supplied with the compiler, or created automatically by the compiler driver as required.

13.7.26 -W*num*

The `-W` option can be used to set the warning level, in the range -9 to 9, or the width of the map file, for values of *num* >= 10.

`-W9` will suppress all warning messages. `-W0` is the default. Setting the warning level to -9 (-W-9) will give the most comprehensive warning messages.

13.7.27 -X

Local symbols can be suppressed from a symbol file with this option. `Global` symbols will always appear in the symbol file.

13.7.28 -Z

Some `local` symbols are compiler generated and not of interest in debugging. This option will suppress from the symbol file all local symbols that have the form of a single alphabetic character, followed by a digit string. The set of letters that can start a trivial symbol is currently "`klfLSu`". The `-Z` option will strip any local symbols starting with one of these letters, and followed by a digit string.

13.8 Invoking the Linker

The linker is called `HLINK`, and normally resides in the `BIN` subdirectory of the compiler installation directory. It may be invoked with no arguments, in which case it will prompt for input from standard input. If the standard input is a file, no prompts will be printed. This manner of invocation is generally useful if the number of arguments to `HLINK` is large. Even if the list of files is too long to fit on one line, continuation lines may be included by leaving a *backslash* `\` at the end of the

preceding line. In this fashion, HLINK commands of almost unlimited length may be issued. For example a link command file called x.lnk and containing the following text:

```
-Z -OX.OBJ -MX.MAP \
-Ptext=0,data=0/,bss,nvram=bss/. \
X.OBJ Y.OBJ Z.OBJ C:\HT-Z80\LIB\Z80-SC.LIB
```

may be passed to the linker by one of the following:

```
hlink @x.lnk
hlink < x.lnk
```

13.9 Map Files

The map file contains information relating to the relocation of psects and the addresses assigned to symbols within those psects. The sections in the map file are as follows; first is a copy of the command line used to invoke the linker. This is followed by the version number of the object code in the first file linked, and the machine type. This is optionally followed by call graph information, depended on the processor and memory model selected. Then are listed all object files that were linked, along with their psect information. Libraries are listed, with each module within the library. The TOTALS section summarises the psects from the object files. The SEGMENTS section summarises major memory groupings. This will typically show RAM and ROM usage. The segment names are derived from the name of the first psect in the segment.

Lastly (not shown in the example) is a symbol table, where each global symbol is listed with its associated psect and link address.

```
Linker command line:
-z -Mmap -pvectors=00h,text,strings,const,im2vecs \
-pbaseram=00h -pramstart=08000h,data/im2vecs,bss/.,stack=09000h \
-pnvram=bss,heap \
-oC:\TEMP\l.obj C:\HT-Z80\LIB\rtz80-s.obj hello.obj \
C:\HT-Z80\LIB\z80-sc.lib
Object code version is 2.4
Machine type is Z80
                Name          Link      Load   Length     Selector
C:\HT-Z80\LIB\rtz80-s.obj
                vectors          0         0       71
                bss           8000      8000       24
                const           FB        FB        1         0
```

```
                text            72        72        82
hello.obj       text            F4        F4         7
C:\HT-Z80\LIB\z80-sc.lib
powerup.obj     vectors         71        71         1
TOTAL           Name          Link      Load    Length
        CLASS   CODE
                vectors          0         0        72
                const           FB        FB         1
                text            72        72        89
        CLASS   DATA
                bss           8000      8000        24
SEGMENTS        Name             Load    Length    Top     Selector
                vectors          000000  0000FC   0000FC        0
                bss              008000  000024   008024     8000
```

13.9.1 Call Graph Information

A call graph is produced for chip types and memory models that use a compiled stack, rather than a hardware stack, to facilitate parameter passing between functions and auto variables defined within a function. When a compiled stack is used, functions are not re-entrant since the function will use a fixed area of memory for its local objects (parameters/auto variables). A function called `foo()`, for example, will use symbols like `?_foo` for parameters and `?a_foo` for auto variables. Compilers such as the PIC, 6805 and V8 use compiled stacks. The 8051 compiler uses a compiled stack in small and medium memory models. The call graph shows information relating to the placement of function parameters and auto variables by the linker. A typical call graph may look something like:

```
Call graph:
*_main size 0,0 offset 0
        _init size 2,3 offset 0
                _ports size 2,2 offset 5
*       _sprintf size 5,10 offset 0
*               _putch
        INDIRECT 4194
                INDIRECT 4194
                        _function_2 size 2,2 offset 0
                        _function size 2,2 offset 5
*_isr->_incr size 2,0 offset 15
```

The graph shows the functions called and the memory usage (RAM) of the functions for their own local objects. In the example above, the symbol `_main` is associated with the function `main()`. It is

shown at the far left of the call graph. This indicates that it is the root of a call tree. The run-time code has the `FNROOT` assembler directive that specifies this. The size field after the name indicates the number of parameters and `auto` variables, respectively. Here, `main()` takes no parameters and defines no `auto` variables. The offset field is the offset at which the function's parameters and auto variables have been placed from the beginning of the area of memory used for this purpose. The run-time code contains a `FNCONF` directive which tells the compiler in which psect parameters and auto variables should reside. This memory will be shown in the map file under the name COMMON.

`Main()` calls a function called `init()`. This function uses a total of two bytes of parameters (it may be two objects of type `char` or one `int`; that is not important) and has three bytes of `auto` variables. These figures are the total of bytes of *memory* consumed by the function. If the function was passed a two-byte `int`, but that was done via a register, then the two bytes would not be included in this total. Since `main()` did not use any of the local object memory, the offset of `init()`'s memory is still at 0.

The function `init()` itself calls another function called `ports()`. This function uses two bytes of parameters and another two bytes of auto variables. Since `ports()` is called by `init()`, its local variables cannot be overlapped with those of `init()`'s, so the offset is 5, which means that `ports()`'s local objects were placed immediately after those of `init()`'s.

The function main also calls `sprintf()`. Since the function `sprintf()` is not active at the same time as `init()` or `ports()`, their local objects can be overlapped and the offset is hence set to 0. `Sprintf()` calls a function `putch()`, but this function uses no memory for parameters (the `char` passed as argument is apparently done so via a register) or locals, so the size and offset are zero and are not printed.

`Main()` also calls another function indirectly using a function pointer. This is indicated by the two `INDIRECT` entries in the graph. The number following is the signature value of functions that could potentially be called by the indirect call. This number is calculated from the parameters and return type of the functions the pointer can indirectly call. The names of any functions that have this signature value are listed underneath the `INDIRECT` entries. Their inclusion does not mean that they were called (there is no way to determine that), but that they could potentially be called.

The last line shows another function whose name is at the far left of the call graph. This implies that this is the root of another call graph tree. This is an interrupt function which is not called by any code, but which is automatically invoked when an enabled interrupt occurs. This interrupt routine calls the function `incr()`, which is shown shorthand in the graph by the `->` symbol followed by the called function's name instead of having that function shown indented on the following line. This is done whenever the calling function does not takes parameters, nor defines any variables.

Those lines in the graph which are starred with `*` are those functions which are on a critical path in terms of RAM usage. For example, in the above, (`main()` is a trivial example) consider the function `sprintf()`. This uses a large amount of local memory and if you could somehow rewrite it so that it used less local memory, it would reduce the entire program's RAM usage. The functions `init()` and `ports()` have had their local memory overlapped with that of `sprintf()`, so

reducing the size of these functions' local memory will have no affect on the program's RAM usage. Their memory usage could be increased, as long as the total size of the memory used by these two functions did not exceed that of `sprintf()`, with no additional memory used by the program. So if you have to reduce the amount of RAM used by the program, look at those functions that are starred.

If, when searching a call graph, you notice that a function's parameter and auto areas have been overlapped (i.e. `?a_foo` was placed at the same address as `?_foo`, for example), then check to make sure that you have actually called the function in your program. If the linker has not seen a function actually called, then it overlaps these areas of memory since that are not needed. This is a consequence of the linker's ability to overlap the local memory areas of functions which are not active at the same time. Once the function is called, unique addresses will be assigned to both the parameters and auto objects.

If you are writing a routine that calls C code from assembler, you will need to include the appropriate assembler directives to ensure that the linker sees the C function being called.

13.10 Librarian

The librarian program, `LIBR`, has the function of combining several object files into a single file known as a library. The purposes of combining several such object modules are several.

- fewer files to link
- faster access
- uses less disk space

In order to make the library concept useful, it is necessary for the linker to treat modules in a library differently from object files. If an object file is specified to the linker, it will be linked into the final linked module. A module in a library, however, will only be linked in if it defines one or more symbols previously known, but not defined, to the linker. Thus modules in a library will be linked only if required. Since the choice of modules to link is made on the first pass of the linker, and the library is searched in a linear fashion, it is possible to order the modules in a library to produce special effects when linking. More will be said about this later.

13.10.1 The Library Format

The modules in a library are basically just concatenated, but at the beginning of a library is maintained a directory of the modules and symbols in the library. Since this directory is smaller than the sum of the modules, the linker is speeded up when searching a library since it need read only the directory and not all the modules on the first pass. On the second pass it need read only those modules which are required, seeking over the others. This all minimises disk I/O when linking.

Table 13.2: Librarian command-line options

Option	Effect
`-P`*`width`*	specify page width
`-W`	Suppress non-fatal errors

Table 13.3: Librarian key letter commands

Key	Meaning
`r`	Replace modules
`d`	Delete modules
`x`	Extract modules
`m`	List modules
`s`	Listmodiules with symbols

It should be noted that the library format is geared exclusively toward object modules, and is not a general purpose archiving mechanism as is used by some other compiler systems. This has the advantage that the format may be optimized toward speeding up the linkage process.

13.10.2 Using the Librarian

The librarian program is called LIBR, and the format of commands to it is as follows:

```
LIBR options k file.lib file.obj ...
```

Interpreting this, `LIBR` is the name of the program, *`options`* is zero or more librarian options which affect the output of the program. *`k`* is a key letter denoting the function requested of the librarian (replacing, extracting or deleting modules, listing modules or symbols), *`file.lib`* is the name of the library file to be operated on, and *`file.obj`* is zero or more object file names.

The librarian options are listed in Table 13.2.

The key letters are listed in Table 13.3.

When replacing or extracting modules, the *`file.obj`* arguments are the names of the modules to be replaced or extracted. If no such arguments are supplied, all the modules in the library will be replaced or extracted respectively. Adding a file to a library is performed by requesting the librarian to replace it in the library. Since it is not present, the module will be appended to the library. If the `r` key is used and the library does not exist, it will be created.

Under the `d` key letter, the named object files will be deleted from the library. In this instance, it is an error not to give any object file names.

The `m` and `s` key letters will list the named modules and, in the case of the `s` keyletter, the symbols defined or referenced within (global symbols only are handled by the librarian). As with the `r` and `x` key letters, an empty list of modules means all the modules in the library.

13.10.3 Examples

Here are some examples of usage of the librarian. The following lists the global symbols in the modules `a.obj`, `b.obj` and `c.obj`:

```
LIBR s file.lib a.obj b.obj c.obj
```

This command deletes the object modules `a.obj`, `b.obj` and `c.obj` from the library file.lib:

```
LIBR d file.lib a.obj b.obj c.obj
```

13.10.4 Supplying Arguments

Since it is often necessary to supply many object file arguments to `LIBR`, and command lines are restricted to 127 characters by CP/M and MS-DOS, `LIBR` will accept commands from standard input if no command line arguments are given. If the standard input is attached to the console, `LIBR` will prompt for input. Multiple line input may be given by using a *backslash* as a continuation character on the end of a line. If standard input is redirected from a file, `LIBR` will take input from the file, without prompting. For example:

```
libr
libr> r file.lib 1.obj 2.obj 3.obj \
libr> 4.obj 5.obj 6.obj
```

will perform much the same as if the object files had been typed on the command line. The libr> prompts were printed by `LIBR` itself, the remainder of the text was typed as input.

```
libr <lib.cmd
```

`LIBR` will read input from `lib.cmd`, and execute the command found therein. This allows a virtually unlimited length command to be given to `LIBR`.

13.10.5 Listing Format

A request to LIBR to list module names will simply produce a list of names, one per line, on standard output. The s keyletter will produce the same, with a list of symbols after each module name. Each symbol will be preceded by the letter D or U, representing a definition or reference to the symbol respectively. The -P option may be used to determine the width of the paper for this operation. For example:

```
LIBR -P80 s file.lib
```

will list all modules in file.lib with their global symbols, with the output formatted for an 80 column printer or display.

13.10.6 Ordering of Libraries

The librarian creates libraries with the modules in the order in which they were given on the command line. When updating a library the order of the modules is preserved. Any new modules added to a library after it has been created will be appended to the end.

The ordering of the modules in a library is significant to the linker. If a library contains a module which references a symbol defined in another module in the same library, the module defining the symbol should come after the module referencing the symbol.

13.10.7 Error Messages

LIBR issues various error messages, most of which represent a fatal error, while some represent a harmless occurrence which will nonetheless be reported unless the -W option was used. In this case all warning messages will be suppressed.

13.11 Objtohex

The HI-TECH linker is capable of producing simple binary files, or object files as output. Any other format required must be produced by running the utility program OBJTOHEX. This allows conversion of object files as produced by the linker into a variety of different formats, including various hex formats. The program is invoked thus:

```
OBJTOHEX options inputfile outputfile
```

All of the arguments are optional. If *outputfile* is omitted it defaults to l.hex or l.bin depending on whether the -b option is used. The *inputfile* defaults to l.obj.

The options for OBJTOHEX are listed in Table 13.4. Where an address is required, the format is the same as for HLINK.

Table 13.4: `OBJTOHEX` command-line options

Option	Meaning
`-8`	Produce a CP/M-86 output file
`-A`	Produce an ATDOS `.atx` output file
`-B`*`base`*	Produce a binary file with offset of *`base`*. Default file name is `l.obj`
`-C`*`ckfile`*	Read a list of checksum specifications from *`ckfile`* or standard input
`-D`	Produce a COD file
`-E`	Produce an MS-DOS `.exe` file
`-F`*`fill`*	Fill unused memory with words of value *`fill`* - default value is 0FFh
`-I`	Produce an *Intel* HEX file with linear addressed extended records.
`-L`	Pass relocation information into the output file (used with `.exe` files)
`-M`	Produce a *Motorola* HEX file (S19, S28 or S37 format)
`-N`	Produce an output file for Minix
`-P`*`stk`*	Produce an output file for an *Atari* ST, with optional stack size
`-R`	Include relocation information in the output file
`-S`*`file`*	Write a symbol file into *file*
`-T`	Produce a *Tektronix* HEX file.
-TE	Produce an extended TekHEX file.
`-U`	Produce a COFF output file
`-UB`	Produce a UBROF format file
`-V`	Reverse the order of words and long words in the output file
`-`*`n,m`*	Format either Motorola or Intel HEX file, where *`n`* is the maximum number of bytes per record and *`m`* specifies the record size rounding. Non-rounded records are zero padded to a multiple of *`m`*. *`m`* itself must be a multiple of 2.

13.11.1 Checksum Specifications

If you are generating a HEX file output, please refer to the hexmate section 13.14 for calculating checksums. For OBJTOHEX, the checksum specification allows automated checksum calculation and takes the form of several lines, each line describing one checksum. The syntax of a checksum line is:

```
addr1-addr2 where1-where2 +offset
```

All of *addr1*, *addr2*, *where1*, *where2* and *offset* are hex numbers, without the usual H suffix. Such a specification says that the bytes at *addr1* through to *addr2* inclusive should be summed and the sum placed in the locations *where1* through *where2* inclusive. For an 8 bit checksum these two addresses should be the same. For a checksum stored low byte first, where1 should be less than where2, and vice versa. The *+offset* is optional, but if supplied, the value offset will be used to initialise the checksum. Otherwise it is initialised to zero. For example:

```
0005-1FFF 3-4 +1FFF
```

This will sum the bytes in 5 through 1FFFH inclusive, then add 1FFFH to the sum. The 16 bit checksum will be placed in locations 3 and 4, low byte in 3. The checksum is initialised with 1FFFH to provide protection against an all zero ROM, or a ROM misplaced in memory. A run time check of this checksum would add the last address of the ROM being checksummed into the checksum. For the ROM in question, this should be 1FFFH. The initialization value may, however, be used in any desired fashion.

13.12 Cref

The cross reference list utility CREF is used to format raw cross-reference information produced by the compiler or the assembler into a sorted listing. A raw cross-reference file is produced with the --CR option to the compiler. The assembler will generate a raw cross-reference file with a -C option (most assemblers) or by using an OPT CRE directive (6800 series assemblers) or a XREF control line (PIC assembler). The general form of the CREF command is:

```
cref options files
```

where *options* is zero or more options as described below and *files* is one or more raw cross-reference files. CREF takes the options listed in Table 13.5.

Each option is described in more detail in the following paragraphs.

Table 13.5: `CREF` command-line options

Option	Meaning
`-F`*`prefix`*	Exclude symbols from files with a pathname or filename starting with *`prefix`*
`-H`*`heading`*	Specify a heading for the listing file
`-L`*`len`*	Specify the page length for the listing file
`-O`*`outfile`*	Specify the name of the listing file
`-P`*`width`*	Set the listing width
`-S`*`stoplist`*	Read file *`stoplist`* and ignore any symbols listed.
`-X`*`prefix`*	Exclude and symbols starting with *`prefix`*

13.12.1 -F*prefix*

It is often desired to exclude from the cross-reference listing any symbols defined in a system header file, e.g. `<stdio.h>`. The `-F` option allows specification of a path name prefix that will be used to exclude any symbols defined in a file whose path name begins with that prefix. For example, `-F\` will exclude any symbols from all files with a path name starting with `\`.

13.12.2 -H*heading*

The `-H` option takes a string as an argument which will be used as a header in the listing. The default heading is the name of the first raw cross-ref information file specified.

13.12.3 -L*len*

Specify the length of the paper on which the listing is to be produced, e.g. if the listing is to be printed on 55 line paper you would use a `-L55` option. The default is 66 lines.

13.12.4 -O*outfile*

Allows specification of the output file name. By default the listing will be written to the standard output and may be redirected in the usual manner. Alternatively *`outfile`* may be specified as the output file name.

13.12.5 -P*width*

This option allows the specification of the width to which the listing is to be formatted, e.g. `-P132` will format the listing for a 132 column printer. The default is 80 columns.

13.12.6 -S*stoplist*

The `-S` option should have as its argument the name of a file containing a list of symbols not to be listed in the cross-reference. Multiple stoplists may be supplied with multiple `-S` options.

13.12.7 -X*prefix*

The `-X` option allows the exclusion of symbols from the listing, based on a prefix given as argument to `-X`. For example if it was desired to exclude all symbols starting with the character sequence `xyz` then the option `-Xxyz` would be used. If a digit appears in the character sequence then this will match any digit in the symbol, e.g. `-XX0` would exclude any symbols starting with the letter `X` followed by a digit.

`CREF` will accept wildcard filenames and I/O redirection. Long command lines may be supplied by invoking `CREF` with no arguments and typing the command line in response to the `cref>` prompt. A *backslash* at the end of the line will be interpreted to mean that more command lines follow.

13.13 Cromwell

The `CROMWELL` utility converts code and symbol files into different formats. The formats available are shown in Table 13.6.

The general form of the `CROMWELL` command is:

`CROMWELL` *`options input_files -okey output_file`*

where *`options`* can be any of the options shown in Table 13.7. *`Output_file`* (optional) is the name of the output file. The *`input_files`* are typically the HEX and SYM file. `CROMWELL` automatically searches for the SDB files and reads those if they are found. The options are further described in the following paragraphs.

13.13.1 -Pname

The `-P` options takes a string which is the name of the processor used. `CROMWELL` may use this in the generation of the output format selected.

Table 13.6: `CROMWELL` format types

Key	Format
`cod`	*Bytecraft* COD file
`coff`	COFF file format
`elf`	ELF/DWARF file
`eomf51`	Extended OMF-51 format
`hitech`	HI-TECH Software format
`icoff`	ICOFF file format
`ihex`	*Intel* HEX file format
`omf51`	OMF-51 file format
`pe`	P&E file format
`s19`	*Motorola* HEX file format

Table 13.7: `CROMWELL` command-line options

Option	Description
`-P`*`name`*	Processor name
`-D`	Dump input file
`-C`	Identify input files only
`-F`	Fake local symbols as global
`-O`*`key`*	Set the output format
`-I`*`key`*	Set the input format
`-L`	List the available formats
`-E`	Strip file extensions
`-B`	Specify big-endian byte ordering
`-M`	Strip underscore character
`-V`	Verbose mode

13.13.2 -D

The `-D` option is used to display to the screen details about the named input file in a readable format. The input file can be one of the file types as shown in Table 13.6.

13.13.3 -C

This option will attempt to identify if the specified input files are one of the formats as shown in Table 13.6. If the file is recognised, a confirmation of its type will be displayed.

13.13.4 -F

When generating a COD file, this option can be used to force all local symbols to be represented as global symbols. The may be useful where an emulator cannot read local symbol information from the COD file.

13.13.5 -Okey

This option specifies the format of the output file. The *key* can be any of the types listed in Table 13.6.

13.13.6 -Ikey

This option can be used to specify the default input file format. The *key* can be any of the types listed in Table 13.6.

13.13.7 -L

Use this option to show what file format types are supported. A list similar to that given in Table 13.6 will be shown.

13.13.8 -E

Use this option to tell `CROMWELL` to ignore any filename extensions that were given. The default extension will be used instead.

13.13.9 -B

In formats that support different endian types, use this option to specify big-endian byte ordering.

13.13.10 -M

When generating COD files this option will remove the preceding *underscore* character from symbols.

13.13.11 -V

Turns on verbose mode which will display information about operations `CROMWELL` is performing.

13.14 Hexmate

The Hexmate utility is a program designed to manipulate Intel HEX files. Hexmate is a post-link stage utility that provides the facility to:

- Calculate and store variable-length checksum values
- Fill unused memory locations with known data sequences
- Merge multiple Intel hex files into one output file
- Convert INHX32 files to other INHX formats (eg. INHX8M)
- Detect specific or partial opcode sequences within a hex file
- Find/replace specific or partial opcode sequences
- Provide a map of addresses used in a hex file
- Change or fix the length of data records in a hex file.
- Validate checksums within Intel hex files.

Typical applications for hexmate might include:

- Merging a bootloader or debug module into a main application at build time
- Calculating a checksum over a range of program memory and storing its value in program memory or EEPROM
- Filling unused memory locations with an instruction to send the PC to a known location if it gets lost.
- Storage of a serial number at a fixed address.

Table 13.8: Hexmate command-line options

Option	Effect
-CK	Calculate and store a checksum value
-FILL	Program unused locations with a known value
-FIND	Search and notify if a particular code sequence is detected
-FIND...,REPLACE	Replace the code sequence with a new code sequence
-FORMAT	Specify maximum data record length or select INHX variant
-HELP	Show all options or display help message for specific option
-LOGFILE	Save hexmate analysis of output and various results to a file
-O*file*	Specify the name of the output file
-SERIAL	Store a serial number or code sequence at a fixed address
-STRING	Store an ASCII string at a fixed address
-W	Adjust warning sensitivity
+	Prefix to any option to overwrite other data in its address range if necessary

- Storage of a string (eg. time stamp) at a fixed address.
- Store initial values at a particular memory address (eg. initialise EEPROM)
- Detecting usage of a buggy/restricted instruction
- Adjusting hex records to a fixed length as required by some bootloaders

13.14.1 Hexmate Command Line Options

Some of these hexmate operations may be possible from the compiler's command line driver. However, if hexmate is to be run directly, its usage is:
`hexmate` *<file1.hex ... fileN.hex> <options>*
Where file1.hex through to fileN.hex are a list of input Intel hex files to merge using hexmate. Additional options can be provided to further customize this process. Table 13.8 lists the command line options that hexmate accepts.

The input parameters to hexmate are now discussed in greater detail.

filename.hex A list of INHX32 or INHX8M input files to feed to hexmate. A range restriction can be applied by appending `,startAddress-endAddress`. The data can be stored at an offset address by appending `+offset`. For example, `myfile.hex,-0-1FF+1E00` will read in code

from `myfile.hex` which falls within address range `0h - 1FFh` (inclusive), but write this code to addresses `1E00h - 1FFFh`. Be careful when shifting sections of executable code. Program code shouldn't be shifted unless it can be guarenteed that no part of the program relies upon the absolute location of this code segment.

13.14.1.1 + Prefix

When the `+` operator precedes a parameter or input file, the data obtained from that parameter will be forced into the output file and will overwrite other data existing within its address range. For example, `+input.hex +-STRING@1000="My string"`. Ordinarily, hexmate will issue an error if two sources try to store differing data at the same location. Using the `+` operator informs hexmate that if more than one data source tries to store data to the same address, the one specified with a '+' will take priority.

13.14.1.2 -CK

-CK is for calculating a checksum. The usage of this option is:
-CK=*start-end@destination*[+*offset*][w*Width*][t*Code*] where:
Start and *End* specify the address range that the checksum will be calculated over.
Destination is the address where to store the checksum result. This value cannot be within the range of calculation.
Offset is an optional initial value to add to the checksum result.
Width is optional and specifies the byte-width of the checksum result. Results can be calculated for byte-widths of 1 to 4 bytes. If a positive width is requested, the result will be stored in big-endian byte order. A negative width will cause the result to be stored in little-endian byte order. If the width is left unspecified, the result will be 2 bytes wide and stored in little-endian byte order.
Code is a hexadecimal code that will trail each byte in the checksum result. This can allow each byte of the checksum result to be embedded within an instruction.
For example, `-CK=0-1FFF@2FFE+2100w2` will calculate a checksum over the range `0-1FFFh` and program the checksum result at address `2FFEh`, checksum value will apply an initial offset of `2100h`. The result will be two bytes wide.

13.14.1.3 -FILL

-FILL is used for filling unused memory locations with a known value. The usage of this option is:
-FILL=*Code@Start-End* where:
Code is the opcode that will be programmed to unused locations in memory. Multi-byte codes should be entered in little endian order.
Start and *End* specify the address range that this fill will apply to.

For example, `-FILL=3412@0-1FFF` will program opcode `1234h` in all unused addresses from program memory address `0` to `1FFFh` (Note the endianism). `-FILL` accepts whole bytes of hexadecimal data from 1 to 8 bytes in length.

13.14.1.4 -FIND

This option is used to detect and log occurances of an opcode or partial code sequence. The usage of this option is:
-FIND=*Findcode*[m*Mask*]@*Start-End*[/*Align*][w][t"*Title*"] where:
Findcode is the hexadecimal code sequence to search for and is entered in little endian byte order.
Mask is optional. It allows a bitmask over the Findcode value and is entered in little endian byte order.
Start and *End* limit the address range to search through.
Align is optional. It specifies that a code sequence can only match if it begins on an address which is a multiple of this value.
w, if present will cause hexmate to issue a warning whenever the code sequence is detected.
Title is optional. It allows a title to be given to this code sequence. Defining a title will make log-reports and messages more descriptive and more readable. A title will not affect the actual search results.

TUTORIAL

Let's look at some examples. The option `-FIND=3412@0-7FFF/2w` will detect the code sequence `1234h` when aligned on a `2` (two) byte address boundary, between `0h` and `7FFFh`. `w` indicates that a warning will be issued each time this sequence is found.
Another example, `-FIND=3412M0F00@0-7FFF/2wt"ADDXY"` is same as last example but the code sequence being matched is masked with `000Fh`, so hexmate will search for `123xh`. If a byte-mask is used, is must be of equal byte-width to the opcode it is applied to. Any messaging or reports generated by hexmate will refer to this opcode by the name, *ADDXY* as this was the title defined for this search.

If hexmate is generating a logfile, it will contain the results of all searches. `-FIND` accepts whole bytes of hex data from 1 to 8 bytes in length. Optionally, `-FIND` can be used in conjunction with `,REPLACE` (described below).

13.14.1.5 -FIND...,REPLACE

REPLACE Can only be used in conjunction with a -FIND option. Code sequences that matched the -FIND criteria can be replaced or partially replaced with new codes. The usage for this sub-option

is:
-FIND...,REPLACE=*Code*[m*Mask*] where:
Code is a little endian hexadecimal code to replace the sequences that match the -FIND criteria.
Mask is an optional bitmask to specify which bits within *Code* will replace the code sequence that has been matched. This may be useful if, for example, it is only necessary to modify 4 bits within a 16-bit instruction. The remaining 12 bits can masked and be left unchanged.

13.14.1.6 -FORMAT

-FORMAT can be used to specify a particular variant of INHX format or adjust maximum record length. The usage of this option is:
-FORMAT=*Type*[,*Length*] where:
Type specifies a particular INHX format to generate.
Length is optional and sets the maximum number of bytes per data record. A valid length is between 1 and 16, with 16 being the default.

TUTORIAL

Consider this case. A bootloader trying to download an INHX32 file fails succeed because it cannot process the extended address records which are part of the INHX32 standard. You know that this bootloader can only program data addressed within the range 0 to 64k, and that any data in the hex file outside of this range can be safely disregarded. In this case, by generating the hex file in INHX8M format the operation might succeed. The hexmate option to do this would be `-FORMAT=INHX8M`.
Now consider this. What if the same bootloader also required every data record to contain eight bytes of data, no more, no less? This is possible by combining `-FORMAT` with `-FILL`. Appropriate use of `-FILL` can ensure that there are no gaps in the data for the address range being programmed. This will satisfy the minimum data length requirement. To set the maximum length of data records to eight bytes, just modify the previous option to become `-FORMAT=INHX8M,8`.

The possible types that are supported by this option are listed in Table 13.9. Note that INHX032 is not an actual INHX format. Selection of this type generates an INHX32 file but will also initialize the upper address information to zero. This is a requirement of some device programmers.

13.14.1.7 -HELP

Using -HELP will list all hexmate options. By entering another hexmate option as a parameter of -HELP will show a detailed help message for the given option. For example, `-HELP=string` will show additional help for the `-STRING` hexmate option.

Table 13.9: INHX types used in -FORMAT option

Type	Description
`INHX8M`	Cannot program addresses beyond 64K.
`INHX32`	Can program addresses beyond 64K with extended linear address records.
`INHX032`	INHX32 with initialization of upper address to zero.

13.14.1.8 -LOGFILE

-LOGFILE saves hexfile statistics to the named file. For example, `-LOGFILE=output.log` will analyse the hex file that hexmate is generating and save a report to a file named *output.log*.

13.14.1.9 -O*file*

The generated Intel hex output will be created in this file. For example, `-Oprogram.hex` will save the resultant output to *program.hex*. The output file can take the same name as one of its input files, but by doing so, it will replace the input file entirely.

13.14.1.10 -SERIAL

Store a particular hex value at a fixed address. The usage of this option is:
-SERIAL=*Code*[+/-*Increment*]@*Address*[+/-*Interval*][r*Repetitions*] where:
Code is a hexadecimal value to store and is entered in little endian byte order.
Increment is optional and allows the value of *Code* to change by this value with each repetition (if requested).
Address is the location to store this code, or the first repetition thereof.
Interval is optional and specifies the address shift per repetition of this code.
Repetitions is optional and specifies the number of times to repeat this code.
For example, `-SERIAL=000001@EFFE` will store hex code `00001h` to address `EFFEh`.
Another example, `-SERIAL=0000+2@1000+10r5` will store 5 codes, beginning with value `0000` at address `1000h`. Subsequent codes will appear at address intervals of `+10h` and the code value will change in increments of `+2h`.

13.14.1.11 -STRING

The -STRING option will embed an ASCII string at a fixed address. The usage of this option is:
-STRING@*Address*[t*Code*]=”*Text*” where:
Address is the location to store this string.

Code is optional and allows a byte sequence to trail each byte in the string. This can allow the bytes of the string to be encoded within an instruction.
Text is the string to convert to ASCII and embed.
For example `-STRING@1000="My favourite string"` will store the ASCII data for the string, `My favourite string` (including null terminator) at address `1000h`.
Another example, `-STRING@1000t34="My favourite string"` will store the same string with every byte in the string being trailed with the hexcode `34h`.

Appendix A

Library Functions

The functions within the standard compiler library are listed in this chapter. Each entry begins with the name of the function. This is followed by information analysed into the following headings.

Synopsis This is the C definition of the function, and the header file in which it is declared.

Description This is a narrative description of the function and its purpose.

Example This is an example of the use of the function. It is usually a complete small program that illustrates the function.

Data types If any special data types (structures etc.) are defined for use with the function, they are listed here with their C definition. These data types will be defined in the header file given under heading — Synopsis.

See also This refers you to any allied functions.

Return value The type and nature of the return value of the function, if any, is given. Information on error returns is also included Only those headings which are relevant to each function are used.

ABS

Synopsis

```
#include <stdlib.h>

int abs (int j)
```

Description

The **abs()** function returns the absolute value of **j**.

Example

```
#include <stdio.h>
#include <stdlib.h>

void
main (void)
{
    int a = -5;

    printf("The absolute value of %d is %d\n", a, abs(a));
}
```

Return Value

The absolute value of **j**.

ACOS

Synopsis

```
#include <math.h>

double acos (double f)
```

Description

The **acos()** function implements the converse of cos(), i.e. it is passed a value in the range -1 to +1, and returns an angle in radians whose cosine is equal to that value.

Example

```
#include <math.h>
#include <stdio.h>

/* Print acos() values for -1 to 1 in degrees. */

void
main (void)
{
    float i, a;

    for(i = -1.0; i < 1.0 ; i += 0.1) {
        a = acos(i)*180.0/3.141592;
        printf("acos(%f) = %f degrees\n", i, a);
    }
}
```

See Also

sin(), cos(), tan(), asin(), atan(), atan2()

Return Value

An angle in radians, in the range 0 to π

ASCTIME

Synopsis

```
#include <time.h>

char * asctime (struct tm * t)
```

Description

The **asctime()** function takes the time broken down into the **struct tm** structure, pointed to by its argument, and returns a 26 character string describing the current date and time in the format:

Sun Sep 16 01:03:52 1973\n\0

Note the *newline* at the end of the string. The width of each field in the string is fixed. The example gets the current time, converts it to a **struct tm** pointer with localtime(), it then converts this to ASCII and prints it. The time() function will need to be provided by the user (see time() for details).

Example

```
#include <stdio.h>
#include <time.h>

void
main (void)
{
    time_t clock;
    struct tm * tp;

    time(&clock);
    tp = localtime(&clock);
    printf("%s", asctime(tp));
}
```

See Also

ctime(), gmtime(), localtime(), time()

Return Value

A pointer to the string.

Note

The example will require the user to provide the time() routine as it cannot be supplied with the compiler. See time() for more details.

ASIN

Synopsis

```
#include <math.h>

double asin (double f)
```

Description

The **asin()** function implements the converse of sin(), i.e. it is passed a value in the range -1 to +1, and returns an angle in radians whose sine is equal to that value.

Example

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    float i, a;

    for(i = -1.0; i < 1.0 ; i += 0.1) {
        a = asin(i)*180.0/3.141592;
        printf("asin(%f) = %f degrees\n", i, a);
    }
}
```

See Also

sin(), cos(), tan(), acos(), atan(), atan2()

Return Value

An angle in radians, in the range - π

ASSERT

Synopsis

```
#include <assert.h>

void assert (int e)
```

Description

This macro is used for debugging purposes; the basic method of usage is to place assertions liberally throughout your code at points where correct operation of the code depends upon certain conditions being true initially. An **assert()** routine may be used to ensure at run time that an assumption holds true. For example, the following statement asserts that the pointer tp is not equal to NULL:

assert(tp);

If at run time the expression evaluates to false, the program will abort with a message identifying the source file and line number of the assertion, and the expression used as an argument to it. A fuller discussion of the uses of **assert()** is impossible in limited space, but it is closely linked to methods of proving program correctness.

Example

```
void
ptrfunc (struct xyz * tp)
{
    assert(tp != 0);
}
```

Note

When required for ROM based systems, the underlying routine _fassert(...) will need to be implemented by the user.

ATAN

Synopsis

```
#include <math.h>

double atan (double x)
```

Description

This function returns the arc tangent of its argument, i.e. it returns an angle e in the range - π

Example

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    printf("%f\n", atan(1.5));
}
```

See Also

sin(), cos(), tan(), asin(), acos(), atan2()

Return Value

The arc tangent of its argument.

ATOF

Synopsis

```
#include <stdlib.h>

double atof (const char * s)
```

Description

The **atof()** function scans the character string passed to it, skipping leading blanks. It then converts an ASCII representation of a number to a double. The number may be in decimal, normal floating point or scientific notation.

Example

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    char buf[80];
    double i;

    gets(buf);
    i = atof(buf);
    printf("Read %s: converted to %f\n", buf, i);
}
```

See Also

atoi(), atol()

Return Value

A double precision floating point number. If no number is found in the string, 0.0 will be returned.

ATOI

Synopsis

```
#include <stdlib.h>

int atoi (const char * s)
```

Description

The **atoi()** function scans the character string passed to it, skipping leading blanks and reading an optional sign. It then converts an ASCII representation of a decimal number to an integer.

Example

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    char buf[80];
    int i;

    gets(buf);
    i = atoi(buf);
    printf("Read %s: converted to %d\n", buf, i);
}
```

See Also

xtoi(), atof(), atol()

Return Value

A signed integer. If no number is found in the string, 0 will be returned.

ATOL

Synopsis

```
#include <stdlib.h>

long atol (const char * s)
```

Description

The **atol()** function scans the character string passed to it, skipping leading blanks. It then converts an ASCII representation of a decimal number to a long integer.

Example

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    char buf[80];
    long i;

    gets(buf);
    i = atol(buf);
    printf("Read %s: converted to %ld\n", buf, i);
}
```

See Also

atoi(), atof()

Return Value

A long integer. If no number is found in the string, 0 will be returned.

BSEARCH

Synopsis

```
#include <stdlib.h>

void * bsearch (const void * key, void * base, size_t n_memb,
size_t size, int (*compar)(const void *, const void *))
```

Description

The **bsearch()** function searches a sorted array for an element matching a particular key. It uses a binary search algorithm, calling the function pointed to by **compar** to compare elements in the array.

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

struct value {
    char name[40];
    int value;
} values[100];

int
val_cmp (const void * p1, const void * p2)
{
    return strcmp(((const struct value *)p1)->name,
        ((const struct value *)p2)->name);
}

void
main (void)
{
    char inbuf[80];
    int i;
    struct value * vp;
```

```
        i = 0;
        while(gets(inbuf)) {
            sscanf(inbuf,"%s %d", values[i].name, &values[i].value);
            i++;
        }
        qsort(values, i, sizeof values[0], val_cmp);
        vp = bsearch("fred", values, i, sizeof values[0], val_cmp);
        if(!vp)
            printf("Item 'fred' was not found\n");
        else
            printf("Item 'fred' has value %d\n", vp->value);
    }
```

See Also

qsort()

Return Value

A pointer to the matched array element (if there is more than one matching element, any of these may be returned). If no match is found, a null pointer is returned.

Note

The comparison function must have the correct prototype.

CEIL

Synopsis

```
#include <math.h>

double ceil (double f)
```

Description

This routine returns the smallest whole number not less than **f**.

Example

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    double j;

    scanf("%lf", &j);
    printf("The ceiling of %lf is %lf\n", j, ceil(j));
}
```

CGETS

Synopsis

```
#include <conio.h>

char * cgets (char * s)
```

Description

The **cgets()** function will read one line of input from the console into the buffer passed as an argument. It does so by repeated calls to getche(). As characters are read, they are buffered, with *backspace* deleting the previously typed character, and *ctrl-U* deleting the entire line typed so far. Other characters are placed in the buffer, with a *carriage return* or *line feed (newline)* terminating the function. The collected string is null terminated.

Example

```
#include <conio.h>
#include <string.h>

char buffer[80];

void
main (void)
{
    for(;;) {
        cgets(buffer);
        if(strcmp(buffer, "exit") == 0)
            break;
        cputs("Type 'exit' to finish\n");
    }
}
```

See Also

getch(), getche(), putch(), cputs()

Return Value

The return value is the character pointer passed as the sole argument.

COS

Synopsis

```
#include <math.h>

double cos (double f)
```

Description

This function yields the cosine of its argument, which is an angle in radians. The cosine is calculated by expansion of a polynomial series approximation.

Example

```
#include <math.h>
#include <stdio.h>

#define C 3.141592/180.0

void
main (void)
{
    double i;

    for(i = 0 ; i <= 180.0 ; i += 10)
        printf("sin(%3.0f) = %f, cos = %f\n", i, sin(i*C), cos(i*C));
}
```

See Also

sin(), tan(), asin(), acos(), atan(), atan2()

Return Value

A double in the range -1 to +1.

COSH, SINH, TANH

Synopsis

```
#include <math.h>

double cosh (double f)
double sinh (double f)
double tanh (double f)
```

Description

These functions are the hyperbolic implementations of the trigonometric functions; cos(), sin() and tan().

Example

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    printf("%f\n", cosh(1.5));
    printf("%f\n", sinh(1.5));
    printf("%f\n", tanh(1.5));
}
```

Return Value

The function **cosh()** returns the hyperbolic cosine value.
The function **sinh()** returns the hyperbolic sine value.
The function **tanh()** returns the hyperbolic tangent value.

CPUTS

Synopsis

```
#include <conio.h>

void cputs (const char * s)
```

Description

The **cputs()** function writes its argument string to the console, outputting *carriage returns* before each *newline* in the string. It calls putch() repeatedly. On a hosted system **cputs()** differs from puts() in that it reads the console directly, rather than using file I/O. In an embedded system **cputs()** and puts() are equivalent.

Example

```
#include <conio.h>
#include <string.h>

char buffer[80];

void
main (void)
{
    for(;;) {
        cgets(buffer);
        if(strcmp(buffer, "exit") == 0)
            break;
        cputs("Type 'exit' to finish\n");
    }
}
```

See Also

cputs(), puts(), putch()

CTIME

Synopsis

```
#include <time.h>

char * ctime (time_t * t)
```

Description

The **ctime()** function converts the time in seconds pointed to by its argument to a string of the same form as described for asctime(). Thus the example program prints the current time and date.

Example

```
#include <stdio.h>
#include <time.h>

void
main (void)
{
    time_t clock;

    time(&clock);
    printf("%s", ctime(&clock));
}
```

See Also

gmtime(), localtime(), asctime(), time()

Return Value

A pointer to the string.

Note

The example will require the user to provide the time() routine as one cannot be supplied with the compiler. See time() for more detail.

DI, EI

Synopsis

```
#include <intrpt.h>

void ei (void)
void di (void)
```

Description

The **ei()** and **di()** routines enable and disable interrupts respectively. These are implemented as macros defined in **intrpt.h**. On most processors they will expand to an in-line assembler instruction that sets or clears the interrupt enable or mask bit.

The example shows the use of **ei()** and **di()** around access to a long variable that is modified during an interrupt. If this was not done, it would be possible to return an incorrect value, if the interrupt occurred between accesses to successive words of the count value.

Example

```
#include <intrpt.h>

long count;

void
interrupt tick (void)
{
    count++;
}

long
getticks (void)
{
    long val;    /* Disable interrupts around access
                    to count, to ensure consistency.*/
    di();
    val = count;
    ei();
```

```
    return val;
}
```

DIV

Synopsis

```
#include <stdlib.h>

div_t div (int numer, int demon)
```

Description

The **div()** function computes the quotient and remainder of the numerator divided by the denominator.

Example

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    div_t x;

    x = div(12345, 66);
    printf("quotient = %d, remainder = %d\n", x.quot, x.rem);
}
```

Return Value

Returns the quotient and remainder into the **div_t** structure.

EVAL_POLY

Synopsis

```
#include <math.h>

double eval_poly (double x, const double * d, int n)
```

Description

The **eval_poly()** function evaluates a polynomial, whose coefficients are contained in the array **d**, at **x**, for example:

```
y = x*x*d2 + x*d1 + d0.
```

The order of the polynomial is passed in **n**.

Example

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    double x, y;
    double d[3] = {1.1, 3.5, 2.7};

    x = 2.2;
    y = eval_poly(x, d, 2);
    printf("The polynomial evaluated at %f is %f\n", x, y);
}
```

Return Value

A double value, being the polynomial evaluated at **x**.

EXP

Synopsis

```
#include <math.h>

double exp (double f)
```

Description

The **exp()** routine returns the exponential function of its argument, i.e. e to the power of **f**.

Example

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double f;

    for(f = 0.0 ; f <= 5 ; f += 1.0)
        printf("e to %1.0f = %f\n", f, exp(f));
}
```

See Also

log(), log10(), pow()

FABS

Synopsis

```
#include <math.h>

double fabs (double f)
```

Description

This routine returns the absolute value of its double argument.

Example

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    printf("%f %f\n", fabs(1.5), fabs(-1.5));
}
```

See Also

abs()

FLOOR

Synopsis

```
#include <math.h>

double floor (double f)
```

Description

This routine returns the largest whole number not greater than **f**.

Example

```
#include <stdio.h>
#include <math.h>

void
main (void)
{
    printf("%f\n", floor( 1.5 ));
    printf("%f\n", floor( -1.5));
}
```

FREXP

Synopsis

```
#include <math.h>

double frexp (double f, int * p)
```

Description

The **frexp()** function breaks a floating point number into a normalized fraction and an integral power of 2. The integer is stored into the **int** object pointed to by **p**. Its return value x is in the interval (0.5, 1.0) or zero, and **f** equals x times 2 raised to the power stored in ***p**. If **f** is zero, both parts of the result are zero.

Example

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double f;
    int i;

    f = frexp(23456.34, &i);
    printf("23456.34 = %f * 2^%d\n", f, i);
}
```

See Also

ldexp()

GETCH, GETCHE

Synopsis

```
#include <conio.h>

char getch (void)
char getche (void)
```

Description

The **getch()** function reads a single character from the console keyboard and returns it without echoing. The **getche()** function is similar but does echo the character typed.

In an embedded system, the source of characters is defined by the particular routines supplied. By default, the library contains a version of **getch()** that will interface to the Lucifer Debugger. The user should supply an appropriate routine if another source is desired, e.g. a serial port.

The module *getch.c* in the SOURCES directory contains model versions of all the console I/O routines. Other modules may also be supplied, e.g. *ser180.c* has routines for the serial port in a Z180.

Example

```
#include <conio.h>

void
main (void)
{
    char c;

    while((c = getche()) != '\n')
        continue;
}
```

See Also

cgets(), cputs(), ungetch()

GETCHAR

Synopsis

```
#include <stdio.h>

int getchar (void)
```

Description

The **getchar()** routine is a getc(stdin) operation. It is a macro defined in **stdio.h**. Note that under normal circumstances **getchar()** will NOT return unless a *carriage return* has been typed on the console. To get a single character immediately from the console, use the function getch().

Example

```
#include <stdio.h>

void
main (void)
{
    int c;

    while((c = getchar()) != EOF)
        putchar(c);
}
```

See Also

getc(), fgetc(), freopen(), fclose()

Note

This routine is not usable in a ROM based system.

GETS

Synopsis

```
#include <stdio.h>

char * gets (char * s)
```

Description

The **gets()** function reads a line from standard input into the buffer at **s**, deleting the *newline* (cf. fgets()). The buffer is null terminated. In an embedded system, **gets()** is equivalent to cgets(), and results in getche() being called repeatedly to get characters. Editing (with *backspace*) is available.

Example

```
#include <stdio.h>

void
main (void)
{
    char buf[80];

    printf("Type a line: ");
    if(gets(buf))
        puts(buf);
}
```

See Also

fgets(), freopen(), puts()

Return Value

It returns its argument, or NULL on end-of-file.

GMTIME

Synopsis

```
#include <time.h>

struct tm * gmtime (time_t * t)
```

Description

This function converts the time pointed to by **t** which is in seconds since 00:00:00 on Jan 1, 1970, into a broken down time stored in a structure as defined in **time.h**. The structure is defined in the 'Data Types' section.

Example

```
#include <stdio.h>
#include <time.h>

void
main (void)
{
    time_t clock;
    struct tm * tp;

    time(&clock);
    tp = gmtime(&clock);
    printf("It's %d in London\n", tp->tm_year+1900);
}
```

See Also

ctime(), asctime(), time(), localtime()

Return Value

Returns a structure of type **tm**.

Note

The example will require the user to provide the time() routine as one cannot be supplied with the compiler. See time() for more detail.

ISALNUM, ISALPHA, ISDIGIT, ISLOWER et. al.

Synopsis

```
#include <ctype.h>

int isalnum (char c)
int isalpha (char c)
int isascii (char c)
int iscntrl (char c)
int isdigit (char c)
int islower (char c)
int isprint (char c)
int isgraph (char c)
int ispunct (char c)
int isspace (char c)
int isupper (char c)
int isxdigit(char c)
```

Description

These macros, defined in **ctype.h**, test the supplied character for membership in one of several overlapping groups of characters. Note that all except **isascii()** are defined for **c**, if **isascii(c)** is true or if c = EOF.

isalnum(c)	c is in 0-9 or a-z or A-Z
isalpha(c)	c is in A-Z or a-z
isascii(c)	c is a 7 bit ascii character
iscntrl(c)	c is a control character
isdigit(c)	c is a decimal digit
islower(c)	c is in a-z
isprint(c)	c is a printing char
isgraph(c)	c is a non-space printable character
ispunct(c)	c is not alphanumeric
isspace(c)	c is a space, tab or newline
isupper(c)	c is in A-Z
isxdigit(c)	c is in 0-9 or a-f or A-F

Example

```
#include <ctype.h>
#include <stdio.h>

void
main (void)
{
    char buf[80];
    int i;

    gets(buf);
    i = 0;
    while(isalnum(buf[i]))
        i++;
    buf[i] = 0;
    printf("'%s' is the word\n", buf);
}
```

See Also

toupper(), tolower(), toascii()

LDEXP

Synopsis

```
#include <math.h>

double ldexp (double f, int i)
```

Description

The **ldexp()** function performs the inverse of frexp() operation; the integer **i** is added to the exponent of the floating point **f** and the resultant returned.

Example

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double f;

    f = ldexp(1.0, 10);
    printf("1.0 * 2^10 = %f\n", f);
}
```

See Also

frexp()

Return Value

The return value is the integer **i** added to the exponent of the floating point value **f**.

LDIV

Synopsis

```
#include <stdlib.h>

ldiv_t ldiv (long number, long denom)
```

Description

The **ldiv()** routine divides the numerator by the denominator, computing the quotient and the remainder. The sign of the quotient is the same as that of the mathematical quotient. Its absolute value is the largest integer which is less than the absolute value of the mathematical quotient.

The **ldiv()** function is similar to the div() function, the difference being that the arguments and the members of the returned structure are all of type **long int**.

Example

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    ldiv_t lt;

    lt = ldiv(1234567, 12345);
    printf("Quotient = %ld, remainder = %ld\n", lt.quot, lt.rem);
}
```

See Also

div()

Return Value

Returns a structure of type **ldiv_t**

LOCALTIME

Synopsis

```
#include <time.h>

struct tm * localtime (time_t * t)
```

Description

The **localtime()** function converts the time pointed to by **t** which is in seconds since 00:00:00 on Jan 1, 1970, into a broken down time stored in a structure as defined in **time.h**. The routine **localtime()** takes into account the contents of the global integer time_zone. This should contain the number of minutes that the local time zone is *westward* of Greenwich. Since there is no way under MS-DOS of actually predetermining this value, by default **localtime()** will return the same result as **gmtime()**.

Example

```
#include <stdio.h>
#include <time.h>

    char * wday[] = {
        "Sunday", "Monday", "Tuesday", "Wednesday",
        "Thursday", "Friday", "Saturday"
};

void
main (void)
{
    time_t clock;
    struct tm * tp;

    time(&clock);
    tp = localtime(&clock);
    printf("Today is %s\n", wday[tp->tm_wday]);
}
```

See Also

ctime(), asctime(), time()

Return Value

Returns a structure of type **tm**.

Note

The example will require the user to provide the time() routine as one cannot be supplied with the compiler. See time() for more detail.

LOG, LOG10

Synopsis

```
#include <math.h>

double log (double f)
double log10 (double f)
```

Description

The **log()** function returns the natural logarithm of **f**. The function **log10()** returns the logarithm to base 10 of **f**.

Example

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double f;

    for(f = 1.0 ; f <= 10.0 ; f += 1.0)
        printf("log(%1.0f) = %f\n", f, log(f));
}
```

See Also

exp(), pow()

Return Value

Zero if the argument is negative.

LONGJMP

Synopsis

```
#include <setjmp.h>

void longjmp (jmp_buf buf, int val)
```

Description

The **longjmp()** function, in conjunction with setjmp(), provides a mechanism for non-local goto's. To use this facility, setjmp() should be called with a **jmp_buf** argument in some outer level function. The call from setjmp() will return 0.

To return to this level of execution, **lonjmp()** may be called with the same **jmp_buf** argument from an inner level of execution. ***Note*** however that the function which called setjmp() must still be active when **longjmp()** is called. Breach of this rule will cause disaster, due to the use of a stack containing invalid data. The **val** argument to **longjmp()** will be the value apparently returned from the setjmp(). This should normally be non-zero, to distinguish it from the genuine setjmp() call.

Example

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

jmp_buf jb;

void
inner (void)
{
    longjmp(jb, 5);
}

void
main (void)
{
    int i;
```

```
        if(i = setjmp(jb)) {
            printf("setjmp returned %d\n", i);
            exit(0);
        }
        printf("setjmp returned 0 - good\n");
        printf("calling inner...\n");
        inner();
        printf("inner returned - bad!\n");
    }
```

See Also

setjmp()

Return Value

The **longjmp()** routine never returns.

Note

The function which called setjmp() must still be active when **longjmp()** is called. Breach of this rule will cause disaster, due to the use of a stack containing invalid data.

MEMCMP

Synopsis

```
#include <string.h>

int memcmp (const void * s1, const void * s2, size_t n)
```

Description

The **memcmp()** function compares two blocks of memory, of length **n**, and returns a signed value similar to strncmp(). Unlike strncmp() the comparison does not stop on a null character. The ASCII collating sequence is used for the comparison, but the effect of including non-ASCII characters in the memory blocks on the sense of the return value is indeterminate. Testing for equality is always reliable.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    int buf[10], cow[10], i;

    buf[0] = 1;
    buf[2] = 4;
    cow[0] = 1;
    cow[2] = 5;
    buf[1] = 3;
    cow[1] = 3;
    i = memcmp(buf, cow, 3*sizeof(int));
    if(i < 0)
        printf("less than\n");
    else if(i > 0)
        printf("Greater than\n");
    else
```

```
        printf("Equal\n");
}
```

See Also

strncpy(), strncmp(), strchr(), memset(), memchr()

Return Value

Returns negative one, zero or one, depending on whether **s1** points to string which is less than, equal to or greater than the string pointed to by **s2** in the collating sequence.

MODF

Synopsis

```
#include <math.h>

double modf (double value, double * iptr)
```

Description

The **modf()** function splits the argument **value** into integral and fractional parts, each having the same sign as **value**. For example, -3.17 would be split into the intergral part (-3) and the fractional part (-0.17).

The integral part is stored as a double in the object pointed to by **iptr**.

Example

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double i_val, f_val;

    f_val = modf( -3.17, &i_val);
}
```

Return Value

The signed fractional part of **value**.

PERSIST_CHECK, PERSIST_VALIDATE

Synopsis

```
#include <sys.h>

int persist_check (int flag)
void persist_validate (void)
```

Description

The **persist_check()** function is used with non-volatile RAM variables, declared with the persistent qualifier. It tests the nvram area, using a magic number stored in a hidden variable by a previous call to **persist_validate()** and a checksum also calculated by **persist_validate()**. If the magic number and checksum are correct, it returns true (non-zero). If either are incorrect, it returns zero. In this case it will optionally zero out and re-validate the non-volatile RAM area (by calling **persist_validate()**). This is done if the flag argument is true.

The **persist_validate()** routine should be called after each change to a persistent variable. It will set up the magic number and recalculate the checksum.

Example

```
#include <sys.h>
#include <stdio.h>

persistent long reset_count;

void
main (void)
{
    if(!persist_check(1))
        printf("Reset count invalid - zeroed\n");
    else
        printf("Reset number %ld\n", reset_count);
    reset_count++;          /* update count */
    persist_validate();     /* and checksum */
    for(;;)
        continue;       /* sleep until next reset */
```

```
}
```

Return Value

FALSE (zero) if the NV-RAM area is invalid; TRUE (non-zero) if the NVRAM area is valid.

POW

Synopsis

```
#include <math.h>

double pow (double f, double p)
```

Description

The **pow()** function raises its first argument, **f**, to the power **p**.

Example

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double f;

    for(f = 1.0 ; f <= 10.0 ; f += 1.0)
        printf("pow(2, %1.0f) = %f\n", f, pow(2, f));
}
```

See Also

log(), log10(), exp()

Return Value

f to the power of **p**.

PRINTF, VPRINTF

Synopsis

```
#include <stdio.h>

int printf (const char * fmt, ...)

#include <stdio.h>
#include <stdarg.h>

int vprintf (const char * fmt, va_list va_arg)
```

Description

The **printf()** function is a formatted output routine, operating on stdout. There are corresponding routines operating on a given stream (fprintf()) or into a string buffer (sprintf()). The **printf()** routine is passed a format string, followed by a list of zero or more arguments. In the format string are conversion specifications, each of which is used to print out one of the argument list values.

Each conversion specification is of the form **%m.nc** where the percent symbol **%** introduces a conversion, followed by an optional width specification **m**. The **n** specification is an optional precision specification (introduced by the dot) and **c** is a letter specifying the type of the conversion.

A minus sign ('-') preceding **m** indicates left rather than right adjustment of the converted value in the field. Where the field width is larger than required for the conversion, blank padding is performed at the left or right as specified. Where right adjustment of a numeric conversion is specified, and the first digit of **m** is 0, then padding will be performed with zeroes rather than blanks. For integer formats, the precision indicates a minimum number of digits to be output, with leading zeros inserted to make up this number if required.

A hash character (**#**) preceding the width indicates that an alternate format is to be used. The nature of the alternate format is discussed below. Not all formats have alternates. In those cases, the presence of the hash character has no effect.

The floating point formats require that the appropriate floating point library is linked. From within HPD this can be forced by selecting the "Float formats in printf" selection in the options menu. From the command line driver, use the option **-LF**.

If the character ***** is used in place of a decimal constant, e.g. in the format **%*d**, then one integer argument will be taken from the list to provide that value. The types of conversion are:

f

Floating point - **m** is the total width and **n** is the number of digits after the decimal point. If **n** is

omitted it defaults to 6. If the precision is zero, the decimal point will be omitted unless the alternate format is specified.

e

Print the corresponding argument in scientific notation. Otherwise similar to **f**.

g

Use **e** or **f** format, whichever gives maximum precision in minimum width. Any trailing zeros after the decimal point will be removed, and if no digits remain after the decimal point, it will also be removed.

o x X u d

Integer conversion - in radices 8, 16, 16, 10 and 10 respectively. The conversion is signed in the case of **d**, unsigned otherwise. The precision value is the total number of digits to print, and may be used to force leading zeroes. E.g. **%8.4x** will print at least 4 hex digits in an 8 wide field. Preceding the key letter with an **l** indicates that the value argument is a long integer. The letter **X** prints out hexadecimal numbers using the upper case letters *A*-*F* rather than *a*-f as would be printed when using **x**. When the alternate format is specified, a leading zero will be supplied for the octal format, and a leading 0x or 0X for the hex format.

s

Print a string - the value argument is assumed to be a character pointer. At most **n** characters from the string will be printed, in a field **m** characters wide.

c

The argument is assumed to be a single character and is printed literally.

Any other characters used as conversion specifications will be printed. Thus **%** will produce a single percent sign.

The **vprintf()** function is similar to **printf()** but takes a variable argument list pointer rather than a list of arguments. See the description of va_start() for more information on variable argument lists. An example of using **vprintf()** is given below.

Example

```
printf("Total = %4d%", 23)
       yields 'Total =   23%'

printf("Size is %lx" , size)
       where size is a long, prints size
       as hexadecimal.

printf("Name = %.8s", "a1234567890")
       yields 'Name = a1234567'
```

```
printf("xx%*d", 3, 4)
       yields 'xx  4'


/* vprintf example */

#include        <stdio.h>

int
error (char * s, ...)
{
    va_list ap;

    va_start(ap, s);
    printf("Error: ");
    vprintf(s, ap);
    putchar('\n');
    va_end(ap);
}

void
main (void)
{
    int i;

    i = 3;
    error("testing 1 2 %d", i);
}
```

See Also

fprintf(), sprintf()

Return Value

The **printf()** and **vprintf()** functions return the number of characters written to stdout.

PUTCH

Synopsis

```
#include <conio.h>

void putch (char c)
```

Description

The **putch()** function outputs the character **c** to the console screen, prepending a *carriage return* if the character is a *newline*. In a CP/M or MS-DOS system this will use one of the system I/O calls. In an embedded system this routine, and associated others, will be defined in a hardware dependent way. The standard **putch()** routines in the embedded library interface either to a serial port or to the Lucifer Debugger.

Example

```
#include <conio.h>

char * x = "This is a string";

void
main (void)
{
    char * cp;

    cp = x;
    while(*x)
        putch(*x++);
    putch('\n');
}
```

See Also

cgets(), cputs(), getch(), getche()

PUTCHAR

Synopsis

```
#include <stdio.h>

int putchar (int c)
```

Description

The **putchar()** function is a putc() operation on stdout, defined in **stdio.h**.

Example

```
#include <stdio.h>

char * x = "This is a string";

void
main (void)
{
    char * cp;

    cp = x;
    while(*x)
        putchar(*x++);
    putchar('\n');
}
```

See Also

putc(), getc(), freopen(), fclose()

Return Value

The character passed as argument, or EOF if an error occurred.

Note

This routine is not usable in a ROM based system.

PUTS

Synopsis

```
#include <stdio.h>

int puts (const char * s)
```

Description

The **puts()** function writes the string **s** to the *stdout stream*, appending a *newline*. The null character terminating the string is not copied.

Example

```
#include <stdio.h>

void
main (void)
{
    puts("Hello, world!");
}
```

See Also

fputs(), gets(), freopen(), fclose()

Return Value

EOF is returned on error; zero otherwise.

QSORT

Synopsis

```
#include <stdlib.h>

void qsort (void * base, size_t nel, size_t width,
int (*func)(const void *, const void *))
```

Description

The **qsort()** function is an implementation of the quicksort algorithm. It sorts an array of **nel** items, each of length **width** bytes, located contiguously in memory at **base**. The argument **func** is a pointer to a function used by **qsort()** to compare items. It calls **func** with pointers to two items to be compared. If the first item is considered to be greater than, equal to or less than the second then **func** should return a value greater than zero, equal to zero or less than zero respectively.

Example

```
#include <stdio.h>
#include <stdlib.h>

int aray[] = {
    567, 23, 456, 1024, 17, 567, 66
};

int
sortem (const void * p1, const void * p2)
{
    return *(int *)p1 - *(int *)p2;
}

void
main (void)
{
    register int i;
```

```
        qsort(aray, sizeof aray/sizeof aray[0], sizeof aray[0], sortem);
        for(i = 0 ; i != sizeof aray/sizeof aray[0] ; i++)
            printf("%d\t", aray[i]);
        putchar('\n');
    }
```

Note

The function parameter must be a pointer to a function of type similar to:
int func (const void *, const void *)
i.e. it must accept two const void * parameters, and must be prototyped.

RAM_VECTOR, CHANGE_VECTOR, READ_RAM_VECTOR

Synopsis

```
#include <intrpt.h>

void RAM_VECTOR (unsigned vector, isr func)
void CHANGE_VECTOR (unsigned vector, isr func)
void (* READ_RAM_VECTOR (unsigned vector)(void))
```

Description

The **RAM_VECTOR()**, **CHANGE_VECTOR()** and **READ_RAM_VECTOR()** macros are used to initialize, modify and read interrupt vectors which are directed through internal RAM based interrupt vectors. These macros should only be used for vectors which need to be modifiable, so as to point at different interrupt functions at different points in the program. The **CHANGE_VECTOR()** and **READ_RAM_VECTOR()** macros should only be used with interrupt vectors which have been initialized using **RAM_VECTOR()**, otherwise garbage will be returned.

Please refer to the section "*Interrupt Handling in C*" in this manual for further details.

Example

```
volatile unsigned char wait_flag;

interrupt void wait_handler(void)
{
    ++wait_flag;
}

void wait_for_serial_intr(void)
{
    interrupt void (*old_handler)(void);

    di();
    old_handler = READ_RAM_VECTOR(RXI);
    wait_flag = 0;
    CHANGE_VECTOR(RXI, wait_handler);
}
```

See Also

di(), ei(), ROM_VECTOR()

Note

These macros, for the Z80/Z180, may only be used with mode 2 interrupts.

RAND

Synopsis

```
#include <stdlib.h>

int rand (void)
```

Description

The **rand()** function is a pseudo-random number generator. It returns an integer in the range 0 to 32767, which changes in a pseudo-random fashion on each call. The algorithm will produce a deterministic sequence if started from the same point. The starting point is set using the **srand()** call. The example shows use of the **time()** function to generate a different starting point for the sequence each time.

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

void
main (void)
{
    time_t toc;
    int i;

    time(&toc);
    srand((int)toc);
    for(i = 0 ; i != 10 ; i++)
        printf("%d\t", rand());
    putchar('\n');
}
```

See Also

srand()

Note

The example will require the user to provide the time() routine as one cannot be supplied with the compiler. See time() for more detail.

REALLOC

Synopsis

```
#include <stdlib.h>

void * realloc (void * ptr, size_t cnt)
```

Description

The **realloc()** function frees the block of memory at **ptr**, which should have been obtained by a previous call to malloc(), calloc() or **realloc()**, then attempts to allocate **cnt** bytes of dynamic memory, and if successful copies the contents of the block of memory located at **ptr** into the new block.

At most, **realloc()** will copy the number of bytes which were in the old block, but if the new block is smaller, will only copy **cnt** bytes.

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

void
main (void)
{
    char * cp;

    cp = malloc(255);
    if(gets(cp))
        cp = realloc(cp, strlen(cp)+1);
    printf("buffer now %d bytes long\n", strlen(cp)+1);
}
```

See Also

malloc(), calloc()

Return Value

A pointer to the new (or resized) block. NULL if the block could not be expanded. A request to shrink a block will never fail.

ROM_VECTOR

Synopsis

```
#include <intrpt.h>

void ROM_VECTOR (unsigned vector, isr func, unsigned psw)
```

Description

The **ROM_VECTOR()** macro is used to set up a "*hard coded*" ROM vector, which points to an interrupt handler. This macro does not generate any code which is executed at run-time, so it can be placed anywhere in your code. **ROM_VECTOR()** generates in-line assembler code, so the vector address passed to it may be in any format acceptable to the assembler.

Please refer to the section "*Interrupt Handling in C*", in this manual for further details.

See Also

di(), ei(), RAM_VECTOR()

SCANF, VSCANF

Synopsis

```
#include <stdio.h>

int scanf (const char * fmt, ...)

#include <stdio.h>
#include <stdarg.h>

int vscanf (const char *, va_list ap)
```

Description

The **scanf()** function performs formatted input ("de-editing") from the *stdin stream*. Similar functions are available for streams in general, and for strings. The function **vscanf()** is similar, but takes a pointer to an argument list rather than a series of additional arguments. This pointer should have been initialised with va_start().

The input conversions are performed according to the **fmt** string; in general a character in the format string must match a character in the input; however a space character in the format string will match zero or more "white space" characters in the input, i.e. *spaces, tabs or newlines*.

A conversion specification takes the form of the character **%**, optionally followed by an assignment suppression character (’*****’), optionally followed by a numerical maximum field width, followed by a conversion specification character. Each conversion specification, unless it incorporates the assignment suppression character, will assign a value to the variable pointed at by the next argument. Thus if there are two conversion specifications in the **fmt** string, there should be two additional pointer arguments.

The conversion characters are as follows:

o x d

Skip white space, then convert a number in base 8, 16 or 10 radix respectively. If a field width was supplied, take at most that many characters from the input. A leading minus sign will be recognized.

f

Skip white space, then convert a floating number in either conventional or scientific notation. The field width applies as above.

s

Skip white space, then copy a maximal length sequence of non-white-space characters. The pointer

argument must be a pointer to char. The field width will limit the number of characters copied. The resultant string will be null terminated.

c

Copy the next character from the input. The pointer argument is assumed to be a pointer to char. If a field width is specified, then copy that many characters. This differs from the **s** format in that white space does not terminate the character sequence.

The conversion characters **o**, **x**, **u**, **d** and **f** may be preceded by an **l** to indicate that the corresponding pointer argument is a pointer to long or double as appropriate. A preceding **h** will indicate that the pointer argument is a pointer to short rather than int.

Example

```
scanf("%d %s", &a, &c)
     with input " 12s"
     will assign 12 to a, and "s" to s.

scanf("%3cd %lf", &c, &f)
     with input " abcd -3.5"
     will assign " abc" to c, and -3.5 to f.
```

See Also

fscanf(), sscanf(), printf(), va_arg()

Return Value

The **scanf()** function returns the number of successful conversions; EOF is returned if end-of-file was seen before any conversions were performed.

SETJMP

Synopsis

```
#include <setjmp.h>

int setjmp (jmp_buf buf)
```

Description

The **setjmp()** function is used with longjmp() for non-local goto's. See longjmp() for further information.

Example

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

jmp_buf jb;

void
inner (void)
{
    longjmp(jb, 5);
}

void
main (void)
{
    int i;

    if(i = setjmp(jb)) {
        printf("setjmp returned %d\n", i);
        exit(0);
    }
    printf("setjmp returned 0 - good\n");
    printf("calling inner...\n");
```

```
        inner();
        printf("inner returned - bad!\n");
    }
```

See Also

longjmp()

Return Value

The **setjmp()** function returns zero after the real call, and non-zero if it apparently returns after a call to longjmp().

SET_VECTOR

Synopsis

```
#include <intrpt.h>

isr set_vector (isr * vector, isr func)
```

Description

This routine allows an interrupt vector to be initialized. The first argument should be the address of the interrupt vector (not the vector number but the actual address) cast to a pointer to **isr**, which is a typedef'd pointer to an interrupt function. The second argument should be the function which you want the interrupt vector to point to. This must be declared using the **interrupt** type qualifier.

Not all compilers support this routine; the macros ROM_VECTOR(), RAM_VECTOR() and CHANGE_VECTOR() are used with some processors. These routines are to be preferred even where **set_vector()** is supported. See **intrpt.h** or the processor specific manual section to determine what is supported for a particular compiler.

The example shown sets up a vector for the DOS ctrl-BREAK interrupt.

Example

```
#include <signal.h>
#include <stdlib.h>
#include <intrpt.h>

static far interrupt void
brkintr (void)
{
    exit(-1);
}

#define BRKINT  0x23
#define BRKINTV ((far isr *)(BRKINT * 4))

void
set_trap (void)
{
```

```
        set_vector(BRKINTV, brkintr);
    }
```

See Also

di(), ei(), ROM_VECTOR(), RAM_VECTOR(), CHANGE_VECTOR()

Return Value

The return value of **set_vector()** is the previous contents of the vector, if **set_vector()** is implemented as a function. If it is implemented as a macro, it has no return value.

Note

The **set_vector()** routine is equivalent to ROM_VECTOR() and is present only for compatibility with version 5 and 6 HI-TECH compilers. It is suggested that ROM_VECTOR() be used in place of **set_vector()** for maximum compatibility with future versions of HI-TECH C.

SIN

Synopsis

```
#include <math.h>

double sin (double f)
```

Description

This function returns the sine function of its argument.

Example

```
#include <math.h>
#include <stdio.h>

#define C 3.141592/180.0

void
main (void)
{
    double i;

    for(i = 0 ; i <= 180.0 ; i += 10)
        printf("sin(%3.0f) = %f, cos = %f\n", i, sin(i*C), cos(i*C));
}
```

See Also

cos(), tan(), asin(), acos(), atan(), atan2()

Return Value

Sine vale of **f**.

SPRINTF, VSPRINTF

Synopsis

```
#include <stdio.h>

int sprintf (char * buf, const char * fmt, ...)

#include <stdio.h>
#include <stdarg.h>

int vsprintf (char * buf, const char * fmt, va_list ap)
```

Description

The **sprintf()** function operates in a similar fashion to printf(), except that instead of placing the converted output on the *stdout stream*, the characters are placed in the buffer at **buf**. The resultant string will be null terminated, and the number of characters in the buffer will be returned.

The **vsprintf()** function is similar to **sprintf()** but takes a variable argument list pointer rather than a list of arguments. See the description of va_start() for more information on variable argument lists.

See Also

printf(), fprintf(), sscanf()

Return Value

Both these routines return the number of characters placed into the buffer.

SQRT

Synopsis

```
#include <math.h>

double sqrt (double f)
```

Description

The function **sqrt()**, implements a square root routine using Newton's approximation.

Example

```
#include <math.h>
#include <stdio.h>

void
main (void)
{
    double i;

    for(i = 0 ; i <= 20.0 ; i += 1.0)
        printf("square root of %.1f = %f\n", i, sqrt(i));
}
```

See Also

exp()

Return Value

Returns the value of the square root.

Note

A domain error occurs if the argument is negative.

SRAND

Synopsis

```
#include <stdlib.h>

void srand (unsigned int seed)
```

Description

The **srand()** function initializes the random number generator accessed by rand() with the given **seed**. This provides a mechanism for varying the starting point of the pseudo-random sequence yielded by rand(). On the z80, a good place to get a truly random seed is from the refresh register. Otherwise timing a response from the console will do, or just using the system time.

Example

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

void
main (void)
{
    time_t toc;
    int i;

    time(&toc);
    srand((int)toc);
    for(i = 0 ; i != 10 ; i++)
        printf("%d\t", rand());
    putchar('\n');
}
```

See Also

rand()

STRCAT

Synopsis

```
#include <string.h>

char * strcat (char * s1, const char * s2)
```

Description

This function appends (catenates) string **s2** to the end of string **s1**. The result will be null terminated. The argument **s1** must point to a character array big enough to hold the resultant string.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char buffer[256];
    char * s1, * s2;

    strcpy(buffer, "Start of line");
    s1 = buffer;
    s2 = " ... end of line";
    strcat(s1, s2);
    printf("Length = %d\n", strlen(buffer));
    printf("string = \"%s\"\n", buffer);
}
```

See Also

strcpy(), strcmp(), strncat(), strlen()

Return Value

The value of **s1** is returned.

STRCHR, STRICHR

Synopsis

```
#include <string.h>

char * strchr (const char * s, int c)
char * strichr (const char * s, int c)
```

Description

The **strchr()** function searches the string **s** for an occurrence of the character **c**. If one is found, a pointer to that character is returned, otherwise NULL is returned.

The **strichr()** function is the case-insensitive version of this function.

Example

```
#include <strings.h>
#include <stdio.h>

void
main (void)
{
    static char temp[] = "Here it is...";
    char c = ’s’;

    if(strchr(temp, c))
        printf("Character %c was found in string\n", c);
    else
        printf("No character was found in string");
}
```

See Also

strrchr(), strlen(), strcmp()

Return Value

A pointer to the first match found, or NULL if the character does not exist in the string.

Note

Although the function takes an integer argument for the character, only the lower 8 bits of the value are used.

STRCMP, STRICMP

Synopsis

```
#include <string.h>

int strcmp (const char * s1, const char * s2)
int stricmp (const char * s1, const char * s2)
```

Description

The **strcmp()** function compares its two, null terminated, string arguments and returns a signed integer to indicate whether **s1** is less than, equal to or greater than **s2**. The comparison is done with the standard collating sequence, which is that of the ASCII character set.

The **stricmp()** function is the case-insensitive version of this function.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    int i;

    if((i = strcmp("ABC", "ABc")) < 0)
        printf("ABC is less than ABc\n");
    else if(i > 0)
        printf("ABC is greater than ABc\n");
    else
        printf("ABC is equal to ABc\n");
}
```

See Also

strlen(), strncmp(), strcpy(), strcat()

Return Value

A signed integer less than, equal to or greater than zero.

Note

Other C implementations may use a different collating sequence; the return value is negative, zero or positive, i.e. do not test explicitly for negative one (-1) or one (1).

STRCPY

Synopsis

```
#include <string.h>

char * strcpy (char * s1, const char * s2)
```

Description

This function copies a null terminated string **s2** to a character array pointed to by **s1**. The destination array must be large enough to hold the entire string, including the null terminator.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char buffer[256];
    char * s1, * s2;

    strcpy(buffer, "Start of line");
    s1 = buffer;
    s2 = " ... end of line";
    strcat(s1, s2);
    printf("Length = %d\n", strlen(buffer));
    printf("string = \"%s\"\n", buffer);
}
```

See Also

strncpy(), strlen(), strcat(), strlen()

Return Value

The destination buffer pointer **s1** is returned.

STRCSPN

Synopsis

```
#include <string.h>

size_t strcspn (const char * s1, const char * s2)
```

Description

The **strcspn()** function returns the length of the initial segment of the string pointed to by **s1** which consists of characters NOT from the string pointed to by **s2**.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    static char set[] = "xyz";

    printf("%d\n", strcspn( "abcdevwxyz", set));
    printf("%d\n", strcspn( "xxxbcadefs", set));
    printf("%d\n", strcspn( "1234567890", set));
}
```

See Also

strspn()

Return Value

Returns the length of the segment.

STRDUP

Synopsis

```
#include <string.h>

char * strdup (const char * s1)
```

Description

The **strdup()** function returns a pointer to a new string which is a duplicate of the string pointed to by **s1**. The space for the new string is obtained using malloc(). If the new string cannot be created, a null pointer is returned.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    char * ptr;

    ptr = strdup("This is a copy");
    printf("%s\n", ptr);
}
```

Return Value

Pointer to the new string, or NULL if the new string cannot be created.

STRLEN

Synopsis

```
#include <string.h>

size_t strlen (const char * s)
```

Description

The **strlen()** function returns the number of characters in the string **s**, not including the null terminator.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char buffer[256];
    char * s1, * s2;

    strcpy(buffer, "Start of line");
    s1 = buffer;
    s2 = " ... end of line";
    strcat(s1, s2);
    printf("Length = %d\n", strlen(buffer));
    printf("string = \"%s\"\n", buffer);
}
```

Return Value

The number of characters preceding the null terminator.

STRNCAT

Synopsis

```
#include <string.h>

char * strncat (char * s1, const char * s2, size_t n)
```

Description

This function appends (catenates) string **s2** to the end of string **s1**. At most **n** characters will be copied, and the result will be null terminated. **s1** must point to a character array big enough to hold the resultant string.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char buffer[256];
    char * s1, * s2;

    strcpy(buffer, "Start of line");
    s1 = buffer;
    s2 = " ... end of line";
    strncat(s1, s2, 5);
    printf("Length = %d\n", strlen(buffer));
    printf("string = \"%s\"\n", buffer);
}
```

See Also

strcpy(), strcmp(), strcat(), strlen()

Return Value

The value of **s1** is returned.

STRNCMP, STRNICMP

Synopsis

```
#include <string.h>

int strncmp (const char * s1, const char * s2, size_t n)
int strnicmp (const char * s1, const char * s2, size_t n)
```

Description

The **strcmp()** function compares its two, null terminated, string arguments, up to a maximum of **n** characters, and returns a signed integer to indicate whether **s1** is less than, equal to or greater than **s2**. The comparison is done with the standard collating sequence, which is that of the ASCII character set.

The **stricmp()** function is the case-insensitive version of this function.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    int i;

    i = strcmp("abcxyz", "abcxyz");
    if(i == 0)
        printf("Both strings are equal\n");
    else if(i > 0)
        printf("String 2 less than string 1\n");
    else
        printf("String 2 is greater than string 1\n");
}
```

See Also

strlen(), strcmp(), strcpy(), strcat()

Return Value

A signed integer less than, equal to or greater than zero.

Note

Other C implementations may use a different collating sequence; the return value is negative, zero or positive, i.e. do not test explicitly for negative one (-1) or one (1).

STRNCPY

Synopsis

```
#include <string.h>

char * strncpy (char * s1, const char * s2, size_t n)
```

Description

This function copies a null terminated string **s2** to a character array pointed to by **s1**. At most **n** characters are copied. If string **s2** is longer than **n** then the destination string will not be null terminated. The destination array must be large enough to hold the entire string, including the null terminator.

Example

```
#include <string.h>
#include <stdio.h>

void
main (void)
{
    char buffer[256];
    char * s1, * s2;

    strncpy(buffer, "Start of line", 6);
    s1 = buffer;
    s2 = " ... end of line";
    strcat(s1, s2);
    printf("Length = %d\n", strlen(buffer));
    printf("string = \"%s\"\n", buffer);
}
```

See Also

strcpy(), strcat(), strlen(), strcmp()

Return Value

The destination buffer pointer **s1** is returned.

STRPBRK

Synopsis

```
#include <string.h>

char * strpbrk (const char * s1, const char * s2)
```

Description

The **strpbrk()** function returns a pointer to the first occurrence in string **s1** of any character from string **s2**, or a null pointer if no character from **s2** exists in **s1**.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    char * str = "This is a string.";

    while(str != NULL) {
        printf( "%s\n", str );
        str = strpbrk( str+1, "aeiou" );
    }
}
```

Return Value

Pointer to the first matching character, or NULL if no character found.

STRRCHR, STRRICHR

Synopsis

```
#include <string.h>

char * strrchr (char * s, int c)
char * strrichr (char * s, int c)
```

Description

The **strrchr()** function is similar to the **strchr()** function, but searches from the end of the string rather than the beginning, i.e. it locates the *last* occurrence of the character **c** in the null terminated string **s**. If successful it returns a pointer to that occurrence, otherwise it returns NULL.

The **strrichr()** function is the case-insensitive version of this function.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    char * str = "This is a string.";

    while(str != NULL) {
        printf( "%s\n", str );
        str = strrchr( str+1, 's');
    }
}
```

See Also

strchr(), strlen(), strcmp(), strcpy(), strcat()

Return Value

A pointer to the character, or NULL if none is found.

STRSPN

Synopsis

```
#include <string.h>

size_t strspn (const char * s1, const char * s2)
```

Description

The **strspn()** function returns the length of the initial segment of the string pointed to by **s1** which consists entirely of characters from the string pointed to by **s2**.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    printf("%d\n", strspn("This is a string", "This"));
    printf("%d\n", strspn("This is a string", "this"));
}
```

See Also

strcspn()

Return Value

The length of the segment.

STRSTR, STRISTR

Synopsis

```
#include <string.h>

char * strstr (const char * s1, const char * s2)
char * stristr (const char * s1, const char * s2)
```

Description

The **strstr()** function locates the first occurrence of the sequence of characters in the string pointed to by **s2** in the string pointed to by **s1**.

The **stristr()** routine is the case-insensitive version of this function.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    printf("%d\n", strstr("This is a string", "str"));
}
```

Return Value

Pointer to the located string or a null pointer if the string was not found.

STRTOK

Synopsis

```
#include <string.h>

char * strtok (char * s1, const char * s2)
```

Description

A number of calls to **strtok()** breaks the string **s1** (which consists of a sequence of zero or more text tokens separated by one or more characters from the separator string **s2**) into its separate tokens.

The first call must have the string **s1**. This call returns a pointer to the first character of the first token, or NULL if no tokens were found. The inter-token separator character is overwritten by a null character, which terminates the current token.

For subsequent calls to **strtok()**, **s1** should be set to a null pointer. These calls start searching from the end of the last token found, and again return a pointer to the first character of the next token, or NULL if no further tokens were found.

Example

```
#include <stdio.h>
#include <string.h>

void
main (void)
{
    char * ptr;
    char buf[] = "This is a string of words.";
    char * sep_tok = ".,?! ";

    ptr = strtok(buf, sep_tok);
    while(ptr != NULL) {
        printf("%s\n", ptr);
        ptr = strtok(NULL, sep_tok);
    }
}
```

Return Value

Returns a pointer to the first character of a token, or a null pointer if no token was found.

Note

The separator string **s2** may be different from call to call.

TAN

Synopsis

```
#include <math.h>

double tan (double f)
```

Description

The **tan()** function calculates the tangent of **f**.

Example

```
#include <math.h>
#include <stdio.h>

#define C 3.141592/180.0

void
main (void)
{
    double i;

    for(i = 0 ; i <= 180.0 ; i += 10)
        printf("tan(%3.0f) = %f\n", i, tan(i*C));
}
```

See Also

sin(), cos(), asin(), acos(), atan(), atan2()

Return Value

The tangent of **f**.

TIME

Synopsis

```
#include <time.h>

time_t time (time_t * t)
```

Description

This function is not provided as it is dependant on the target system supplying the current time. This function will be user implemented. When implemented, this function should return the current time in seconds since 00:00:00 on Jan 1, 1970. If the argument **t** is not equal to NULL, the same value is stored into the object pointed to by **t**.

Example

```
#include <stdio.h>
#include <time.h>

void
main (void)
{
    time_t clock;

    time(&clock);
    printf("%s", ctime(&clock));
}
```

See Also

ctime(), gmtime(), localtime(), asctime()

Return Value

This routine when implemented will return the current time in seconds since 00:00:00 on Jan 1, 1970.

Note

The **time()** routine is not supplied, if required the user will have to implement this routine to the specifications outlined above.

TOLOWER, TOUPPER, TOASCII

Synopsis

```
#include <ctype.h>

char toupper (int c)
char tolower (int c)
char toascii (int c)
```

Description

The **toupper()** function converts its lower case alphabetic argument to upper case, the **tolower()** routine performs the reverse conversion and the **toascii()** macro returns a result that is guaranteed in the range 0-0177. The functions **toupper()** and **tolower()** return their arguments if it is not an alphabetic character.

Example

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

void
main (void)
{
    char * array1 = "aBcDE";
    int i;

    for(i=0;i < strlen(array1); ++i) {
        printf("%c", tolower(array1[i]));
    }
    printf("\n");
}
```

See Also

islower(), isupper(), isascii(), et. al.

UNGETCH

Synopsis

```
#include <conio.h>

void ungetch (char c)
```

Description

The **ungetch()** function will push back the character **c** onto the console stream, such that a subsequent getch() operation will return the character. At most one level of push back will be allowed.

See Also

getch(), getche()

VA_START, VA_ARG, VA_END

Synopsis

```
#include <stdarg.h>

void va_start (va_list ap, parmN)
type va_arg  (ap, type)
void va_end  (va_list ap)
```

Description

These macros are provided to give access in a portable way to parameters to a function represented in a prototype by the ellipsis symbol (...), where type and number of arguments supplied to the function are not known at compile time.

The rightmost parameter to the function (shown as **parmN**) plays an important role in these macros, as it is the starting point for access to further parameters. In a function taking variable numbers of arguments, a variable of type **va_list** should be declared, then the macro **va_start()** invoked with that variable and the name of **parmN**. This will initialize the variable to allow subsequent calls of the macro **va_arg()** to access successive parameters.

Each call to **va_arg()** requires two arguments; the variable previously defined and a type name which is the type that the next parameter is expected to be. Note that any arguments thus accessed will have been widened by the default conventions to *int*, *unsigned int* or *double*. For example if a character argument has been passed, it should be accessed by **va_arg(ap, int)** since the *char* will have been widened to *int*.

An example is given below of a function taking one integer parameter, followed by a number of other parameters. In this example the function expects the subsequent parameters to be pointers to char, but note that the compiler is not aware of this, and it is the programmers responsibility to ensure that correct arguments are supplied.

Example

```
#include <stdio.h>
#include <stdarg.h>

void
pf (int a, ...)
{
```

```
    va_list ap;

    va_start(ap, a);
    while(a--)
        puts(va_arg(ap, char *));
    va_end(ap);
}

void
main (void)
{
    pf(3, "Line 1", "line 2", "line 3");
}
```

XTOI

Synopsis

```
#include <stdlib.h>

unsigned xtoi (const char * s)
```

Description

The **xtoi()** function scans the character string passed to it, skipping leading blanks reading an optional sign, and converts an ASCII representation of a hexadecimal number to an integer.

Example

```
#include <stdlib.h>
#include <stdio.h>

void
main (void)
{
    char buf[80];
    int i;

    gets(buf);
    i = xtoi(buf);
    printf("Read %s: converted to %x\n", buf, i);
}
```

See Also

atoi()

Return Value

A signed integer. If no number is found in the string, zero will be returned.

Appendix B

Error and Warning Messages

This chapter lists most error, warning and advisory messages from all HI-TECH C compilers, with an explanation of each message. Most messages have been assigned a unique number which appears in brackets before each message in this chapter, and which is also printed by the compiler when the message is issued. The messages shown here are sorted by their number. Un-numbered messages appear toward the end and are sorted alphabetically.

The name of the application(s) that could have produced the messages are listed in brackets opposite the error message. In some cases examples of code or options that could trigger the error are given. The use of * in the error message is used to represent a string that the compiler will substitute that is specific to that particular error.

Note that one problem in your C or assembler source code may trigger more than one error message.

(100) unterminated #if[n][def] block from line * ***(Preprocessor)***

A `#if` or similar block was not terminated with a matching `#endif`, e.g.:

```
#if INPUT        /* error flagged here */
void main(void)
{
  run();
}                /* no #endif was found in this module */
```

(101) #* may not follow #else ***(Preprocessor)***

A `#else` or `#elif` has been used in the same conditional block as a `#else`. These can only follow a `#if`, e.g.:

```
#ifdef FOO
  result = foo;
#else
  result = bar;
#elif defined(NEXT)   /* the #else above terminated the #if */
  result = next(0);
#endif
```

(102) #* must be in an #if ***(Preprocessor)***

The `#elif`, `#else` or `#endif` directive must be preceded by a matching `#if` line. If there is an apparently corresponding `#if` line, check for things like extra `#endif`'s, or improperly terminated comments, e.g.:

```
#ifdef FOO
  result = foo;
#endif
  result = bar;
#elif defined(NEXT)   /* the #endif above terminated the #if */
  result = next(0);
#endif
```

(103) #error: * ***(Preprocessor)***

This is a programmer generated error; there is a directive causing a deliberate error. This is normally used to check compile time defines etc. Remove the directive to remove the error, but first check as to why the directive is there.

(104) preprocessor assertion failure ***(Preprocessor)***

The argument to a preprocessor `#assert` directive has evaluated to zero. This is a programmer induced error.

```
#assert SIZE == 4  /* size should never be 4 */
```

(105) no #asm before #endasm ***(Preprocessor)***

A `#endasm` operator has been encountered, but there was no previous matching `#asm`, e.g.:

```
void cleardog(void)
{
  clrwdt
#endasm  /* this ends the in-line assembler, only where did it begin? */
}
```

(106) nested #asm directive ***(Preprocessor)***

It is not legal to nest `#asm` directives. Check for a missing or misspelt `#endasm` directive, e.g.:

```
#asm
  move r0, #0aah
#asm             ; the previous #asm must be closed before opening another
  sleep
#endasm
```

(107) illegal # directive "*" ***(Preprocessor, Parser)***

The compiler does not understand the `#` directive. It is probably a misspelling of a pre-processor `#` directive, e.g.:

```
#indef DEBUG  /* woops -- that should be #undef DEBUG */
```

(108) #if, #ifdef, or #ifndef without an argument ***(Preprocessor)***

The preprocessor directives `#if`, `#ifdef` and `#ifndef` must have an argument. The argument to `#if` should be an expression, while the argument to `#ifdef` or `#ifndef` should be a single name, e.g.:

```
#if             /* woops -- no argument to check */
  output = 10;
#else
  output = 20;
#endif
```

(109) #include syntax error *(Preprocessor)*

The syntax of the filename argument to `#include` is invalid. The argument to `#include` must be a valid file name, either enclosed in double quotes `""` or angle brackets `< >`. Spaces should not be included, and the closing quote or bracket must be present. There should be nothing else on the line other than comments, e.g.:

```
#include stdio.h  /* woops -- should be: #include <stdio.h> */
```

(110) too many file arguments; usage: cpp [input [output]] *(Preprocessor)*

`CPP` should be invoked with at most two file arguments. Contact HI-TECH Support if the preprocessor is being executed by a compiler driver.

(111) redefining macro "*" *(Preprocessor)*

The macro specified is being redefined, to something different to the original definition. If you want to deliberately redefine a macro, use `#undef` first to remove the original definition, e.g.:

```
#define ONE 1
/* elsewhere: */
#define ONE one  /* Is this correct? It will overwrite the first definition. */
```

(112) #define syntax error *(Preprocessor)*

A macro definition has a syntax error. This could be due to a macro or formal parameter name that does not start with a letter or a missing *closing parenthesis* , `)`, e.g.:

```
#define FOO(a, 2b)  bar(a, 2b)  /* 2b is not to be! */
```

(113) unterminated string in macro body *(Preprocessor, Assembler)*

A macro definition contains a string that lacks a closing quote.

(114) illegal #undef argument *(Preprocessor)*

The argument to `#undef` must be a valid name. It must start with a letter, e.g.:

```
#undef 6YYY  /* this isn't a valid symbol name */
```

(115) recursive macro definition of "*" defined by "*" ***(Preprocessor)***

The named macro has been defined in such a manner that expanding it causes a recursive expansion of itself!

(116) end of file within macro argument from line * ***(Preprocessor)***

A macro argument has not been terminated. This probably means the closing parenthesis has been omitted from a macro invocation. The line number given is the line where the macro argument started, e.g.:

```
#define FUNC(a, b) func(a+b)
FUNC(5, 6;    /* woops -- where is the closing bracket? */
```

(117) misplaced constant in #if ***(Preprocessor)***

A constant in a `#if` expression should only occur in syntactically correct places. This error is most probably caused by omission of an operator, e.g.:

```
#if FOO BAR  /* woops -- did you mean: #if FOO == BAR ? */
```

(118) #if value stack overflow ***(Preprocessor)***

The preprocessor filled up its expression evaluation stack in a `#if` expression. Simplify the expression — it probably contains too many parenthesized subexpressions.

(119) illegal #if line ***(Preprocessor)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(120) operator * in incorrect context ***(Preprocessor)***

An operator has been encountered in a `#if` expression that is incorrectly placed, e.g. two binary operators are not separated by a value, e.g.:

```
#if FOO * % BAR == 4  /* what is "* %" ? */
  #define BIG
#endif
```

(121) expression stack overflow at op "*" ***(Preprocessor)***

Expressions in `#if` lines are evaluated using a stack with a size of 128. It is possible for very complex expressions to overflow this. Simplify the expression.

(122) unbalanced paren's, op is "*" ***(Preprocessor)***

The evaluation of a `#if` expression found mismatched parentheses. Check the expression for correct parenthesisation, e.g.:

```
#if ((A) + (B)  /* woops -- a missing ), I think */
  #define ADDED
#endif
```

(123) misplaced "?" or ":", previous operator is * ***(Preprocessor)***

A colon operator has been encountered in a `#if` expression that does not match up with a corresponding `?` operator, e.g.:

```
#if XXX : YYY  /* did you mean:  #if COND ? XXX : YYY */
```

(124) illegal character "*" in #if ***(Preprocessor)***

There is a character in a `#if` expression that has no business being there. Valid characters are the letters, digits and those comprising the acceptable operators, e.g.:

```
#if `YYY`  /* what are these characters doing here? */
  int m;
#endif
```

(125) illegal character (* decimal) in #if ***(Preprocessor)***

There is a non-printable character in a `#if` expression that has no business being there. Valid characters are the letters, digits and those comprising the acceptable operators, e.g.:

```
#if ^SYYY  /* what is this control characters doing here? */
  int m;
#endif
```

(126) can’t use a string in an #if ***(Preprocessor)***

The preprocessor does not allow the use of strings in `#if` expressions, e.g.:

```
#if MESSAGE > “hello”  /* no string operations allowed by the preprocessor */
#define DEBUG
#endif
```

(127) bad #if ... defined() syntax ***(Preprocessor)***

The `defined()` pseudo-function in a preprocessor expression requires its argument to be a single name. The name must start with a letter and should be enclosed in parentheses, e.g.:

```
#if defined(a&b)   /* woops -- defined expects a name, not an expression */
  input = read();
#endif
```

(128) illegal operator in #if ***(Preprocessor)***

A `#if` expression has an illegal operator. Check for correct syntax, e.g.:

```
#if FOO = 6  /* woops -- should that be: #if FOO == 5 ? */
```

(129) unexpected "\" in #if ***(Preprocessor)***

The *backslash* is incorrect in the `#if` statement, e.g.:

```
#if FOO == \34
  #define BIG
#endif
```

(130) #if sizeof, unknown type "*" ***(Preprocessor)***

An unknown type was used in a preprocessor `sizeof()`. The preprocessor can only evaluate `sizeof()` with basic types, or pointers to basic types, e.g.:

```
#if sizeof(unt) == 2  /* woops -- should be: #if sizeof(int) == 2 */
  i = 0xFFFF;
#endif
```

(131) #if ... sizeof: illegal type combination ***(Preprocessor)***

The preprocessor found an illegal type combination in the argument to `sizeof()` in a `#if` expression, e.g.

```
#if sizeof(short long int) == 2  /* short or long? make up your mind */
  i = 0xFFFF;
#endif
```

(132) #if sizeof() error, no type specified ***(Preprocessor)***

Sizeof() was used in a preprocessor `#if` expression, but no type was specified. The argument to `sizeof()` in a preprocessor expression must be a valid simple type, or pointer to a simple type, e.g.:

```
#if sizeof()  /* woops -- size of what? */
  i = 0;
#endif
```

(133) #if ... sizeof: bug, unknown type code 0x* ***(Preprocessor)***

The preprocessor has made an internal error in evaluating a `sizeof()` expression. Check for a malformed type specifier. This is an internal error. Contact HI-TECH Software technical support with details.

(134) #if ... sizeof() syntax error ***(Preprocessor)***

The preprocessor found a syntax error in the argument to `sizeof`, in a `#if` expression. Probable causes are mismatched parentheses and similar things, e.g.:

```
#if sizeof(int == 2)  /* woops -- should be: #if sizeof(int) == 2 */
  i = 0xFFFF;
#endif
```

(135) #if bug, operand = * ***(Preprocessor)***

The preprocessor has tried to evaluate an expression with an operator it does not understand. This is an internal error. Contact HI-TECH Software technical support with details.

(137) strange character "*" after ## ***(Preprocessor)***

A character has been seen after the token catenation operator `##` that is neither a letter nor a digit. Since the result of this operator must be a legal token, the operands must be tokens containing only letters and digits, e.g.:

```
#define cc(a, b) a ## 'b  /* the ' character will not lead to a valid token */
```

(138) strange character (*) after ## ***(Preprocessor)***

An unprintable character has been seen after the token catenation operator `##` that is neither a letter nor a digit. Since the result of this operator must be a legal token, the operands must be tokens containing only letters and digits, e.g.:

```
#define cc(a, b) a ## 'b  /* the ' character will not lead to a valid token */
```

(139) EOF in comment ***(Preprocessor)***

End of file was encountered inside a comment. Check for a missing closing comment flag, e.g.:

```
/* Here is the start of a comment. I'm not sure where I end, though
}
```

(140) can't open command file * ***(Driver, Preprocessor, Assembler, Linker)***

The command file specified could not be opened for reading. Confirm the spelling and path of the file specified on the command line, e.g.:

```
picc @communds
```

should that be:

```
picc @commands
```

(141) can't open output file * ***(Preprocessor, Assembler)***

An output file could not be created. Confirm the spelling and path of the file specified on the command line.

(142) can't open input file * *(Preprocessor, Assembler)*

An input file could not be opened. Confirm the spelling and path of the file specified on the command line.

(144) too many nested #if statements *(Preprocessor)*

`#if`, `#ifdef` etc. blocks may only be nested to a maximum of 32.

(145) cannot open include file "*" *(Preprocessor)*

The named preprocessor include file could not be opened for reading by the preprocessor. Check the spelling of the filename. If it is a standard header file, not in the current directory, then the name should be enclosed in angle brackets <> not quotes. For files not in the current working directory or the standard compiler include directory, you may need to specify an additional include file path to the command-line driver, see Section 10.4.6.

(146) filename work buffer overflow *(Preprocessor)*

A filename constructed while looking for an include file has exceeded the length of an internal buffer. Since this buffer is 4096 bytes long, this is unlikely to happen.

(147) too many include directories *(Preprocessor)*

A maximum of 7 directories may be specified for the preprocessor to search for include files. The number of directories specified with the driver is too great.

(148) too many arguments for macro *(Preprocessor)*

A macro may only have up to 31 parameters, as per the C Standard.

(149) macro work area overflow *(Preprocessor)*

The total length of a macro expansion has exceeded the size of an internal table. This table is normally 8192 bytes long. Thus any macro expansion must not expand into a total of more than 8K bytes.

(150) bug: illegal __ macro "*" *(Preprocessor)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(151) too many arguments in macro expansion ***(Preprocessor)***

There were too many arguments supplied in a macro invocation. The maximum number allowed is 31.

(152) bad dp/nargs in openpar: c = * ***(Preprocessor)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(153) out of space in macro "*" arg expansion ***(Preprocessor)***

A macro argument has exceeded the length of an internal buffer. This buffer is normally 4096 bytes long.

(155) work buffer overflow doing * ## ***(Preprocessor)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(156) work buffer overflow: * ***(Preprocessor)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(157) out of memory ***(Code Generator, Assembler, Optimiser)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(158) invalid disable: * ***(Preprocessor)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(159) too much pushback ***(Preprocessor)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(160) too many errors ***(Preprocessor, Parser, Code Generator, Assembler, Linker)***

There were so many errors that the compiler has given up. Correct the first few errors and many of the later ones will probably go away.

(161) control line "*" within macro expansion *(Preprocessor)*

A preprocessor control line (one starting with a #) has been encountered while expanding a macro. This should not happen.

(163) unexpected text in #control line ignored *(Preprocessor)*

This warning occurs when extra characters appear on the end of a control line, e.g. The extra text will be ignored, but a warning is issued. It is preferable (and in accordance with Standard C) to enclose the text as a comment, e.g.:

```
#if defined(END)
  #define NEXT
#endif END     /* END would be better in a comment here */
```

(164) included file * was converted to lower case *(Preprocessor)*

The file specified to be included was not found, but a file with a lowercase version of the name of the file specified was found and used instead, e.g.:

```
#include "STDIO.H"  /* is this meant to be stdio.h ? */
```

(164) included file * was converted to lower case *(Preprocessor)*

The `#include` file name had to be converted to lowercase before it could be opened.

```
#include <STDIO.H>  /* woops -- should be: #include <stdio.h> */
```

(166) -S, too few values specified in * *(Preprocessor)*

The list of values to the preprocessor (CPP) `-S` option is incomplete. This should not happen if the preprocessor is being invoked by the compiler driver. The values passes to this option represent the sizes of `char`, `short`, `int`, `long`, `float` and `double` types.

(167) -S, too many values, "*" unused *(Preprocessor)*

There were too many values supplied to the -S preprocessor option. See the Error Message `-s, too few values specified in *` on page 388.

(168) unknown option "*" *(Hexmate, Preprocessor)*

This option to the preprocessor/hexmate is not recognized.

(169) strange character after # (*) *(Preprocessor)*

There is an unexpected character after `#`.

(170) symbol "*" not defined in #undef *(Preprocessor)*

The symbol supplied as argument to `#undef` was not already defined. This warning may be disabled with some compilers. This warning can be avoided with code like:

```
#ifdef SYM
  #undef SYM  /* only undefine if defined */
#endif
```

(171) wrong number of macro arguments for "*" - * instead of * *(Preprocessor)*

A macro has been invoked with the wrong number of arguments, e.g.:

```
#define ADD(a, b) (a+b)
ADD(1, 2, 3)            /* woops -- only two arguments required */
```

(172) formal parameter expected after # *(Preprocessor)*

The stringization operator `#` (not to be confused with the leading `#` used for preprocessor control lines) must be followed by a formal macro parameter, e.g.:

```
#define str(x) #y  /* woops -- did you mean x instead of y? */
```

If you need to stringize a token, you will need to define a special macro to do it, e.g.

```
#define __mkstr__(x) #x
```

then use `__mkstr__(token)` wherever you need to convert a token into a string.

(173) undefined symbol "*" in #if, 0 used *(Preprocessor)*

A symbol on a `#if` expression was not a defined preprocessor macro. For the purposes of this expression, its value has been taken as zero. This warning may be disabled with some compilers. Example:

```
#if FOO+BAR    /* e.g. FOO was never #defined */
  #define GOOD
#endif
```

(174) multi-byte constant "*" isn't portable ***(Preprocessor)***

Multi-byte constants are not portable, and in fact will be rejected by later passes of the compiler, e.g.:

```
#if CHAR == 'ab'
  #define MULTI
#endif
```

(175) division by zero in #if, zero result assumed ***(Preprocessor)***

Inside a `#if` expression, there is a division by zero which has been treated as yielding zero, e.g.:

```
#if foo/0  /* divide by 0: was this what you were intending? */
  int a;
#endif
```

(176) missing newline ***(Preprocessor)***

A new line is missing at the end of the line. Each line, including the last line, must have a new line at the end. This problem is normally introduced by editors.

(177) macro "*" wasn't defined ***(Preprocessor)***

A macro name specified in a `-U` option to the preprocessor was not initially defined, and thus cannot be undefined.

(179) nested comments ***(Preprocessor)***

This warning is issued when nested comments are found. A nested comment may indicate that a previous closing comment marker is missing or malformed, e.g.:

```
output = 0;  /* a comment that was left unterminated
flag = TRUE; /* another comment: hey, where did this line go? */
```

(180) unterminated comment in included file ***(Preprocessor)***

Comments begun inside an included file must end inside the included file.

(181) non-scalar types can't be converted *(Parser)*

You can't convert a structure, union or array to another type, e.g.:

```
struct TEST test;
struct TEST * sp;
sp = test;       /* woops -- did you mean: sp = &test; ? */
```

(182) illegal conversion *(Parser)*

This expression implies a conversion between incompatible types, e.g. a conversion of a structure type into an integer, e.g.:

```
struct LAYOUT layout;
int i;
layout = i;          /* an int cannot be converted into a struct */
```

Note that even if a structure only contains an `int`, for example, it cannot be assigned to an `int` variable, and vice versa.

(183) function or function pointer required *(Parser)*

Only a function or function pointer can be the subject of a function call, e.g.:

```
int a, b, c, d;
a = b(c+d);     /* b is not a function -- did you mean a = b*(c+d) ? */
```

(184) can't call an interrupt function *(Parser)*

A function qualified `interrupt` can't be called from other functions. It can only be called by a hardware (or software) interrupt. This is because an `interrupt` function has special function entry and exit code that is appropriate only for calling from an interrupt. An `interrupt` function can call other non-`interrupt` functions.

(185) function does not take arguments *(Parser, Code Generator)*

This function has no parameters, but it is called here with one or more arguments, e.g.:

```
int get_value(void);
void main(void)
{
```

```
  int input;
  input = get_value(6);  /* woops -- the parameter should not be here */
}
```

(186) too many arguments ***(Parser)***

This function does not accept as many arguments as there are here.

```
void add(int a, int b);
add(5, 7, input);              /* this call has too many arguments */
```

(187) too few arguments ***(Parser)***

This function requires more arguments than are provided in this call, e.g.:

```
void add(int a, int b);
add(5);                  /* this call needs more arguments */
```

(188) constant expression required ***(Parser)***

In this context an expression is required that can be evaluated to a constant at compile time, e.g.:

```
int a;
switch(input) {
  case a:       /* woops -- you cannot use a variable as part of a case label */
    input++;
}
```

(189) illegal type for array dimension ***(Parser)***

An array dimension must be either an integral type or an enumerated value.

```
int array[12.5];  /* woops -- twelve and a half elements, eh? */
```

(190) illegal type for index expression ***(Parser)***

An index expression must be either integral or an enumerated value, e.g.:

```
int i, array[10];
i = array[3.5];   /* woops -- exactly which element do you mean? */
```

(191) cast type must be scalar or void ***(Parser)***

A typecast (an abstract type declarator enclosed in parentheses) must denote a type which is either scalar (i.e. not an array or a structure) or the type `void`, e.g.:

```
lip = (long [])input;  /* woops -- maybe: lip = (long *)input */
```

(192) undefined identifier: * ***(Parser)***

This symbol has been used in the program, but has not been defined or declared. Check for spelling errors if you think it has been defined.

(193) not a variable identifier: * ***(Parser)***

This identifier is not a variable; it may be some other kind of object, e.g. a label.

(194)) expected ***(Parser)***

A *closing parenthesis*, `)`, was expected here. This may indicate you have left out this character in an expression, or you have some other syntax error. The error is flagged on the line at which the code first starts to make no sense. This may be a statement following the incomplete expression, e.g.:

```
if(a == b  /* the closing parenthesis is missing here */
  b = 0;   /* the error is flagged here */
```

(195) expression syntax ***(Parser)***

This expression is badly formed and cannot be parsed by the compiler, e.g.:

```
a /=% b;  /* woops -- maybe that should be: a /= b; */
```

(196) struct/union required ***(Parser)***

A structure or union identifier is required before a dot `.`, e.g.:

```
int a;
a.b = 9;  /* woops -- a is not a structure */
```

(197) struct/union member expected ***(Parser)***

A structure or union member name must follow a dot (".") or arrow ("->").

(198) undefined struct/union: * ***(Parser)***

The specified structure or union tag is undefined, e.g.

```
struct WHAT what;  /* a definition for WHAT was never seen */
```

(199) logical type required ***(Parser)***

The expression used as an operand to `if`, `while` statements or to boolean operators like `!` and `&&` must be a scalar integral type, e.g.:

```
struct FORMAT format;
if(format)           /* this operand must be a scaler type */
  format.a = 0;
```

(200) can't take address of register variable ***(Parser)***

A variable declared `register` may not have storage allocated for it in memory, and thus it is illegal to attempt to take the address of it by applying the `&` operator, e.g.:

```
int * proc(register int in)
{
  int * ip = &in;      /* woops -- in may not have an address to take */
  return ip;
}
```

(201) can't take this address ***(Parser)***

The expression which was the operand of the `&` operator is not one that denotes memory storage ("an lvalue") and therefore its address can not be defined, e.g.:

```
ip = &8;  /* woops -- you can't take the address of a literal */
```

(202) only lvalues may be assigned to or modified ***(Parser)***

Only an lvalue (i.e. an identifier or expression directly denoting addressable storage) can be assigned to or otherwise modified, e.g.:

```
int array[10];
int * ip;
char c;
array = ip;    /* array is not a variable, it cannot be written to */
```

A typecast does not yield an lvalue, e.g.:

```
(int)c = 1;    /* the contents of c cast to int is only a intermediate value */
```

However you can write this using pointers:

```
*(int *)&c = 1
```

(203) illegal operation on a bit variable ***(Parser)***

Not all operations on `bit` variables are supported. This operation is one of those, e.g.:

```
bit   b;
int * ip;
ip = &b;  /* woops -- cannot take the address of a bit object */
```

(204) void function cannot return value ***(Parser)***

A void function cannot return a value. Any `return` statement should not be followed by an expression, e.g.:

```
void run(void)
{
  step();
  return 1;     /* either run should not be void, or remove the 1 */
}
```

(205) integral type required ***(Parser)***

This operator requires operands that are of integral type only.

(206) illegal use of void expression ***(Parser)***

A `void` expression has no value and therefore you can't use it anywhere an expression with a value is required, e.g. as an operand to an arithmetic operator.

(207) simple type required for * ***(Parser)***

A simple type (i.e. not an array or structure) is required as an operand to this operator.

(208) operands of * not same type ***(Parser)***

The operands of this operator are of different pointer, e.g.:

```
int * ip;
char * cp, * cp2;
cp = flag ? ip : cp2;  /* result of ? : will either be int * or char * */
```

Maybe you meant something like:

```
cp = flag ? (char *)ip : cp2;
```

(209) type conflict ***(Parser)***

The operands of this operator are of incompatible types.

(210) bad size list ***(Parser)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(212) missing number after pragma "pack" ***(Parser)***

The pragma `pack` requires a decimal number as argument. This specifies the alignment of each member within the structure. Use this with caution as some processors enforce alignment and will not operate correctly if word fetches are made on odd boundaries, e.g.:

```
#pragma pack  /* what is the alignment value */
```

Maybe you meant something like:

```
#pragma pack 2
```

(214) missing number after pragma "interrupt_level" ***(Parser)***

The pragma `interrupt_level` requires an argument from 0 to 7.

(215) missing argument to "pragma switch" ***(Parser)***

The pragma switch requires an argument of `auto`, `direct` or `simple`, e.g.:

```
#pragma switch  /* woops -- this requires a switch mode */
```

maybe you meant something like:

```
#pragma switch simple
```

(216) missing argument to "pragma psect" ***(Parser)***

The pragma `psect` requires an argument of the form *`oldname`*`=`*`newname`* where *`oldname`* is an existing psect name known to the compiler, and *`newname`* is the desired new name, e.g.:

```
#pragma psect  /* woops -- this requires an psect to redirect */
```

maybe you meant something like:

```
#pragma psect text=specialtext
```

(218) missing name after pragma "inline" ***(Parser)***

The `inline` pragma expects the name of a function to follow. The function name must be recognized by the code generator for it to be expanded; other functions are not altered, e.g.:

```
#pragma inline  /* what is the function name? */
```

maybe you meant something like:

```
#pragma inline memcpy
```

(219) missing name after pragma "printf_check" ***(Parser)***

The `printf_check` pragma expects the name of a function to follow. This specifies printf-style format string checking for the function, e.g.

```
#pragma printf_check  /* what function is to be checked? */
```

Maybe you meant something like:

```
#pragma printf_check sprintf
```

Pragmas for all the standard printf-like function are already contained in `<stdio.h>`.

(220) exponent expected ***(Parser)***

A floating point constant must have at least one digit after the `e` or `E`., e.g.:

```
float f;
f = 1.234e;  /* woops -- what is the exponent? */
```

(221) hex digit expected ***(Parser)***

After `0x` should follow at least one of the hex digits `0-9` and `A-F` or `a-f`, e.g.:

```
a = 0xg6;  /* woops -- was that meant to be a = 0xf6 ? */
```

(222) binary digit expected ***(Parser)***

A binary digit was expected following the `0b` format specifier, e.g.

```
i = 0bf000;  /* wooops -- f000 is not a base two value */
```

(223) digit out of range ***(Parser, Assembler, Optimiser)***

A digit in this number is out of range of the radix for the number, e.g. using the digit 8 in an octal number, or hex digits A-F in a decimal number. An octal number is denoted by the digit string commencing with a zero, while a hex number starts with "0X" or "0x". For example:

```
int a = 058;  /* a leading 0 implies octal which has digits 0 thru 7 */
```

(225) missing character in character constant ***(Parser)***

The character inside the single quotes is missing, e.g.:

```
char c = ″;  /* the character value of what? */
```

(226) char const too long ***(Parser)***

A character constant enclosed in single quotes may not contain more than one character, e.g.:

```
c = ’12’;  /* woops -- only one character may be specified */
```

(227) "." expected after ".." ***(Parser)***

The only context in which two successive dots may appear is as part of the *ellipsis* symbol, which must have 3 dots. (An *ellipsis* is used in function prototypes to indicate a variable number of parameters.)

Either `..` was meant to be an *ellipsis* symbol which would require you to add an extra dot, or it was meant to be a *structure member operator* which would require you remove one dot.

(228) illegal character (*) ***(Parser)***

This character is illegal in the C code. Valid characters are the letters, digits and those comprising the acceptable operators, e.g.:

```
c = ‘a‘;  /* woops -- did you mean c = ’a’; ? */
```

(229) unknown qualifier "*" given to -A ***(Parser)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(230) missing arg to -A ***(Parser)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(231) unknown qualifier "*" given to -I ***(Parser)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(232) missing arg to -I ***(Parser)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(233) bad -Q option * ***(Parser)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(234) close error (disk space?) ***(Parser)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(236) simple integer expression required ***(Parser)***

A simple integral expression is required after the operator `@`, used to associate an absolute address with a variable, e.g.:

```
int address;
char LOCK @ address;
```

(237) function "*" redefined ***(Parser)***

More than one definition for a function has been encountered in this module. Function overloading is illegal, e.g.:

```
int twice(int a)
{
  return a*2;
}
long twice(long a)  /* only one prototype & definition of rv can exist */
{
  return a*2;
}
```

(238) illegal initialisation ***(Parser)***

You can't initialise a `typedef` declaration, because it does not reserve any storage that can be initialised, e.g.:

```
typedef unsigned int uint = 99;  /* woops -- uint is a type, not a variable */
```

(239) identifier redefined: * (from line *) ***(Parser)***

This identifier has already been defined in the same scope. It cannot be defined again, e.g.:

```
  int a;  /* a filescope variable called "a" */
  int a;  /* this attempts to define another with the same name */
```

Note that variables with the same name, but defined with different scopes are legal, but not recommended.

(240) too many initializers ***(Parser)***

There are too many initializers for this object. Check the number of initializers against the object definition (array or structure), e.g.:

```
int ivals[3] = { 2, 4, 6, 8};  /* three elements, but four initializers */
```

(241) initialization syntax ***(Parser)***

The initialisation of this object is syntactically incorrect. Check for the correct placement and number of braces and commas, e.g.:

```
int iarray[10] = {{'a', 'b', 'c'};  /* woops -- one two many {s */
```

(242) illegal type for switch expression ***(Parser)***

A `switch` operation must have an expression that is either an integral type or an enumerated value, e.g:

```
double d;
switch(d) {  /* woops -- this must be integral */
  case '1.0':
    d = 0;
}
```

(243) inappropriate break/continue ***(Parser)***

A `break` or `continue` statement has been found that is not enclosed in an appropriate control structure. A `continue` can only be used inside a `while`, `for` or `do while` loop, while `break` can only be used inside those loops or a `switch` statement, e.g.:

```
switch(input) {
  case 0:
    if(output == 0)
      input = 0xff;
    }              /* woops -- this shouldn't be here and closed the switch */
    break;         /* this should be inside the switch */
```

(244) default case redefined ***(Parser)***

There is only allowed to be one `default` label in a switch statement. You have more than one, e.g.:

```
switch(a) {
default:     /* if this is the default case... */
  b = 9;
  break;
default:     /* then what is this? */
  b = 10;
  break;
```

(245) "default" not in switch *(Parser)*

A label has been encountered called `default` but it is not enclosed by a `switch` statement. A `default` label is only legal inside the body of a `switch` statement.

If there is a `switch` statement before this `default` label, there may be one too many closing braces in the `switch` code which would prematurely terminate the `switch` statement. See example for Error Message `'case' not in switch` on page 402.

(246) "case" not in switch *(Parser)*

A `case` label has been encountered, but there is no enclosing `switch` statement. A `case` label may only appear inside the body of a `switch` statement.

If there is a `switch` statement before this `case` label, there may be one too many closing braces in the `switch` code which would prematurely terminate the `switch` statement, e.g.:

```
switch(input) {
  case '0':
    count++;
    break;
  case '1':
    if(count>MAX)
      count= 0;
    }          /* woops -- this shouldn't be here */
    break;
  case '2':    /* error flagged here */
```

(247) duplicate label * *(Parser)*

The same name is used for a label more than once in this function. Note that the scope of labels is the entire function, not just the block that encloses a label, e.g.:

```
start:
  if(a > 256)
    goto end;
start:           /* error flagged here */
  if(a == 0)
    goto start;  /* which start label do I jump to? */
```

(248) inappropriate "else" *(Parser)*

An `else` keyword has been encountered that cannot be associated with an `if` statement. This may mean there is a missing brace or other syntactic error, e.g.:

```
/* here is a comment which I have forgotten to close...
if(a > b) {
  c = 0;    /* ... that will be closed here, thus removing the "if" */
else        /* my "if" has been lost */
  c = 0xff;
```

(249) probable missing "}" in previous block *(Parser)*

The compiler has encountered what looks like a function or other declaration, but the preceding function has not been ended with a closing brace. This probably means that a closing brace has been omitted from somewhere in the previous function, although it may well not be the last one, e.g.:

```
void set(char a)
{
  PORTA = a;
                  /* the closing brace was left out here */
void clear(void)  /* error flagged here */
{
  PORTA = 0;
}
```

(251) array dimension redeclared *(Parser)*

An array dimension has been declared as a different non-zero value from its previous declaration. It is acceptable to redeclare the size of an array that was previously declared with a zero dimension, but not otherwise, e.g.:

```
extern int array[5];
int array[10];        /* woops -- has it 5 or 10 elements? */
```

(252) argument * conflicts with prototype *(Parser)*

The argument specified (argument 0 is the left most argument) of this function definition does not agree with a previous prototype for this function, e.g.:

```
extern int calc(int, int);   /* this is supposedly calc's prototype */
int calc(int a, long int b)  /* hmmm -- which is right? */
{                            /* error flagged here */
  return sin(b/a);
}
```

(253) argument list conflicts with prototype *(Parser)*

The argument list in a function definition is not the same as a previous prototype for that function. Check that the number and types of the arguments are all the same.

```
extern int calc(int);   /* this is supposedly calc's prototype */
int calc(int a, int b)  /* hmmm -- which is right? */
{                       /* error flagged here */
  return a + b;
}
```

(254) undefined *: * *(Parser)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(255) not a member of the struct/union * *(Parser)*

This identifier is not a member of the structure or union type with which it used here, e.g.:

```
struct {
  int a, b, c;
} data;
if(data.d)     /* woops -- there is no member d in this structure */
  return;
```

(256) too much indirection *(Parser)*

A pointer declaration may only have 16 levels of indirection.

(257) only register storage class allowed *(Parser)*

The only storage class allowed for a function parameter is `register`, e.g.:

```
void process(static int input)
```

(258) duplicate qualifier ***(Parser)***

There are two occurrences of the same qualifier in this type specification. This can occur either directly or through the use of a typedef. Remove the redundant qualifier. For example:

```
typedef volatile int vint;
volatile vint very_vol;   /* woops -- this results in two volatile qualifiers */
```

(259) can't be both far and near ***(Parser)***

It is illegal to qualify a type as both `far` and `near`, e.g.:

```
far near int spooky;  /* woops -- choose either far or near, not both */
```

(260) undefined enum tag: * ***(Parser)***

This enum tag has not been defined, e.g.:

```
enum WHAT what;  /* a definition for WHAT was never seen */
```

(261) member * redefined ***(Parser)***

This name of this member of the struct or union has already been used in this `struct` or `union`, e.g.:

```
struct {
  int a;
  int b;
  int a;  /* woops -- a different name is required here */
} input;
```

(262) struct/union redefined: * ***(Parser)***

A structure or union has been defined more than once, e.g.:

```
struct {
  int a;
} ms;
struct {
  int a;
} ms;    /* was this meant to be the same name as above? */
```

(263) members cannot be functions ***(Parser)***

A member of a structure or a union may not be a function. It may be a pointer to a function, e.g.:

```
struct {
  int a;
  int get(int);  /* this should be a pointer: int (*get)(int); */
} object;
```

(264) bad bitfield type ***(Parser)***

A bitfield may only have a type of `int` (`signed` or `unsigned`), e.g.:

```
struct FREG {
  char b0:1;   /* woops -- these must be part of an int, not char */
  char   :6;
  char b7:1;
} freg;
```

(265) integer constant expected ***(Parser)***

A *colon* appearing after a member name in a structure declaration indicates that the member is a bitfield. An integral constant must appear after the *colon* to define the number of bits in the bitfield, e.g.:

```
struct {
  unsigned first:  /* woops -- should be: unsigned first; */
  unsigned second;
} my_struct;
```

If this was meant to be a structure with bitfields, then the following illustrates an example:

```
struct {
  unsigned first : 4;  /* 4 bits wide */
  unsigned second: 4;  /* another 4 bits */
} my_struct;
```

(266) storage class illegal ***(Parser)***

A structure or union member may not be given a storage class. Its storage class is determined by the storage class of the structure, e.g.:

```
struct {
  static int first;  /* no additional qualifiers may be present with members */
} ;
```

(267) bad storage class ***(Code Generator)***

The code generator has encounterd a variable definition whose storage class is invalid, e.g.:

```
auto int foo;           /* auto not permitted with global variables */
int power(static int a)  /* paramters may not be static */
{
  return foo * a;
}
```

(268) inconsistent storage class ***(Parser)***

A declaration has conflicting storage classes. Only one storage class should appear in a declaration, e.g.:

```
extern static int where;  /* so is it static or extern? */
```

(269) inconsistent type ***(Parser)***

Only one basic type may appear in a declaration, e.g.:

```
int float if;  /* is it int or float? */
```

(270) can't be register ***(Parser)***

Only function parameters or `auto` variables may be declared using the `register` qualifier, e.g.:

```
register int gi;                /* this cannot be qualified register */
int process(register int input)  /* this is okay */
{
  return input + gi;
}
```

(271) can't be long ***(Parser)***

Only int and float can be qualified with long.

```
long char lc;  /* what? */
```

(272) can't be short ***(Parser)***

Only int can be modified with short, e.g.:

```
short float sf;  /* what? */
```

(273) can't have "signed" and "unsigned" together ***(Parser)***

The type modifiers signed and unsigned cannot be used together in the same declaration, as they have opposite meaning, e.g.:

```
signed unsigned int confused;  /* which is it? signed or unsigned? */
```

(274) can't be unsigned ***(Parser)***

A floating point type cannot be made unsigned, e.g.:

```
unsigned float uf;  /* what? */
```

(275) ... illegal in non-prototype arg list ***(Parser)***

The *ellipsis* symbol may only appear as the last item in a prototyped argument list. It may not appear on its own, nor may it appear after argument names that do not have types, i.e. K&R-style non-prototype function definitions. For example:

```
int kandr(a, b, ...)  /* K&R-style non-prototyped function definition */
  int a, b;
{
```

(276) type specifier required for proto arg ***(Parser)***

A type specifier is required for a prototyped argument. It is not acceptable to just have an identifier.

(277) can't mix proto and non-proto args *(Parser)*

A function declaration can only have all prototyped arguments (i.e. with types inside the parentheses) or all K&R style args (i.e. only names inside the parentheses and the argument types in a declaration list before the start of the function body), e.g.:

```
int plus(int a, b)  /* woops -- a is prototyped, b is not */
int b;
{
  return a + b;
}
```

(278) argument redeclared: * *(Parser)*

The specified argument is declared more than once in the same argument list, e.g.

```
int calc(int a, int a)  /* you cannot have two parameters called "a" */
```

(279) can't initialize arg *(Parser)*

A function argument can't have an initialiser in a declaration. The initialisation of the argument happens when the function is called and a value is provided for the argument by the calling function, e.g.:

```
extern int proc(int a = 9);  /* woops -- a is initialized when proc is called */
```

(280) can't have array of functions *(Parser)*

You can't define an array of functions. You can however define an array of pointers to functions, e.g.:

```
int * farray[]();  /* woops -- should be: int (* farray[])(); */
```

(281) functions can't return functions *(Parser)*

A function cannot return a function. It can return a function pointer. A function returning a pointer to a function could be declared like this: int (* (name()))(). Note the many parentheses that are necessary to make the parts of the declaration bind correctly.

(282) functions can't return arrays ***(Parser)***

A function can return only a scalar (simple) type or a structure. It cannot return an array.

(283) dimension required ***(Parser)***

Only the most significant (i.e. the first) dimension in a multi-dimension array may not be assigned a value. All succeeding dimensions must be present as a constant expression, e.g.:

```
enum { one = 1, two };
int get_element(int array[two][])  /* should be, e.g.: int array[][7] */
{
  return array[1][6];
}
```

(285) no identifier in declaration ***(Parser)***

The identifier is missing in this declaration. This error can also occur where the compiler has been confused by such things as missing closing braces, e.g.:

```
void interrupt(void)  /* what is the name of this function? */
{
}
```

(286) declarator too complex ***(Parser)***

This declarator is too complex for the compiler to handle. Examine the declaration and find a way to simplify it. If the compiler finds it too complex, so will anybody maintaining the code.

(287) can't have an array of bits or a pointer to bit ***(Parser)***

It is not legal to have an array of bits, or a pointer to bit variable, e.g.:

```
bit barray[10];  /* wrong -- no bit arrays */
bit * bp;        /* wrong -- no pointers to bit variables */
```

(288) only functions may be void ***(Parser)***

A variable may not be `void`. Only a function can be `void`, e.g.:

```
int a;
void b;  /* this makes no sense */
```

(289) only functions may be qualified interrupt ***(Parser)***

The qualifier `interrupt` may not be applied to anything except a function, e.g.:

```
interrupt int input;  /* variables cannot be qualified interrupt */
```

(290) illegal function qualifier(s) ***(Parser)***

A qualifier has been applied to a function which makes no sense in this context. Some qualifier only make sense when used with an lvalue, e.g. const or volatile. This may indicate that you have forgotten out a star `*` indicating that the function should return a pointer to a qualified object, e.g.

```
const char ccrv(void)  /* woops -- did you mean const * char ccrv(void) ? */
{                      /* error flagged here */
  return ccip;
}
```

(291) not an argument: * ***(Parser)***

This identifier that has appeared in a K&R stype argument declarator is not listed inside the parentheses after the function name, e.g.:

```
int process(input)
int unput;         /* woops -- that should be int input; */
{
}
```

(292) a parameter may not be a function ***(Parser)***

A function parameter may not be a function. It may be a pointer to a function, so perhaps a "*" has been omitted from the declaration.

(293) bad size in index_type ***(Parser)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(294) can't allocate * bytes of memory ***(Code Generator, Hexmate)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(295) expression too complex ***(Parser)***

This expression has caused overflow of the compiler's internal stack and should be re-arranged or split into two expressions.

(297) bad arg (*) to tysize ***(Parser)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(298) EOF in #asm ***(Preprocessor)***

An end of file has been encountered inside a `#asm` block. This probably means the `#endasm` is missing or misspelt, e.g.:

```
#asm
  mov  r0, #55
  mov  [r1], r0
}                /* woops -- where is the #endasm */
```

(300) unexpected EOF ***(Parser)***

An end-of-file in a C module was encountered unexpectedly, e.g.:

```
void main(void)
{
  init();
  run();    /* is that it? What about the close brace */
```

(301) EOF on string file ***(Parser)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(302) can't reopen * ***(Parser)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(303) no memory for string buffer ***(Parser)***

The parser was unable to allocate memory for the longest string encountered, as it attempts to sort and merge strings. Try reducing the number or length of strings in this module.

(305) can't open * ***(Code Generator, Assembler, Optimiser, Cromwell)***

An input file could not be opened. Confirm the spelling and path of the file specified on the command line.

(306) out of far memory ***(Code Generator)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(307) too many qualifier names ***(Parser)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(308) too many cases in switch ***(Code Generator)***

There are too many `case` labels in this `switch` statement. The maximum allowable number of `case` labels in any one `switch` statement is 511.

(309) too many symbols ***(Assembler)***

There are too many symbols for the assembler's symbol table. Reduce the number of symbols in your program.

(310)] expected ***(Parser)***

A closing square bracket was expected in an array declaration or an expression using an array index, e.g.

```
process(carray[idx);  /* woops -- should be: process(carray[idx]); */
```

(313) function body expected ***(Parser)***

Where a function declaration is encountered with K&R style arguments (i.e. argument names but no types inside the parentheses) a function body is expected to follow, e.g.:

```
int get_value(a, b);  /* the function block must follow, not a semicolon */
```

(314) ; expected ***(Parser)***

A *semicolon* is missing from a statement. A close brace or keyword was found following a statement with no terminating *semicolon*, e.g.:

```
while(a) {
  b = a--  /* woops -- where is the semicolon? */
}          /* error is flagged here */
```

Note: Omitting a semicolon from statements not preceeding a close brace or keyword typically results in some other error being issed for the following code which the parser assums to be part of the original statement.

(315) { expected ***(Parser)***

An *opening brace* was expected here. This error may be the result of a function definition missing the *opening brace*, e.g.:

```
void process(char c)      /* woops -- no opening brace after the prototype */
  return max(c, 10) * 2;  /* error flagged here */
}
```

(316) } expected ***(Parser)***

A *closing brace* was expected here. This error may be the result of a initialized array missing the *closing brace*, e.g.:

```
char carray[4] = { 1, 2, 3, 4;  /* woops -- no closing brace */
```

(317) (expected ***(Parser)***

An *opening parenthesis*, `(`, was expected here. This must be the first token after a `while`, `for`, `if`, `do` or `asm` keyword, e.g.:

```
if a == b  /* should be: if(a == b) */
  b = 0;
```

(318) string expected ***(Parser)***

The operand to an `asm` statement must be a string enclosed in parentheses, e.g.:

```
asm(nop);  /* that should be asm("nop");
```

(319) while expected ***(Parser)***

The keyword `while` is expected at the end of a `do` statement, e.g.:

```
do {
  func(i++);
}             /* do the block while what condition is true? */
if(i > 5)     /* error flagged here */
  end();
```

(320) : expected ***(Parser)***

A *colon* is missing after a `case` label, or after the keyword `default`. This often occurs when a *semicolon* is accidentally typed instead of a *colon*, e.g.:

```
switch(input) {
  case 0;         /* woops -- that should have been: case 0: */
    state = NEW;
```

(321) label identifier expected ***(Parser)***

An identifier denoting a label must appear after `goto`, e.g.:

```
if(a)
  goto 20;  /* this is not BASIC -- a valid C label must follow a goto */
```

(322) enum tag or { expected ***(Parser)***

After the keyword `enum` must come either an identifier that is or will be defined as an `enum` tag, or an opening brace, e.g.:

```
enum 1, 2;  /* should be, e.g.: enum {one=1, two }; */
```

(323) struct/union tag or "{" expected ***(Parser)***

An identifier denoting a structure or union or an opening brace must follow a `struct` or `union` keyword, e.g.:

```
struct int a;  /* this is not how you define a structure */
```

You might mean something like:

```
struct {
  int a;
} my_struct;
```

(324) too many arguments for format string ***(Parser)***

There are too many arguments for this format string. This is harmless, but may represent an incorrect format string, e.g.:

```
printf("%d - %d", low, high, median);  /* woops -- missed a placeholder? */
```

(325) error in format string ***(Parser)***

There is an error in the format string here. The string has been interpreted as a `printf()` style format string, and it is not syntactically correct. If not corrected, this will cause unexpected behaviour at run time, e.g.:

```
printf("%l", lll);  /* woops -- maybe: printf("%ld", lll); */
```

(326) long argument required ***(Parser)***

A long argument is required for this format specifier. Check the number and order of format specifiers and corresponding arguments, e.g.:

```
printf("%lx", 2);  /* woops -- maybe you meant: printf("%lx", 2L);
```

(328) integral argument required ***(Parser)***

An integral argument is required for this printf-style format specifier. Check the number and order of format specifiers and corresponding arguments, e.g.:

```
printf("%d", 1.23);  /* woops -- either wrong number or wrong placeholder */
```

(329) double float argument required ***(Parser)***

The printf format specifier corresponding to this argument is `%f` or similar, and requires a floating point expression. Check for missing or extra format specifiers or arguments to printf.

```
printf("%f", 44);  /* should be: printf("%f", 44.0); */
```

(330) pointer to * argument required ***(Parser)***

A pointer argument is required for this format specifier. Check the number and order of format specifiers and corresponding arguments.

(331) too few arguments for format string ***(Parser)***

There are too few arguments for this format string. This would result in a garbage value being printed or converted at run time, e.g.:

```
printf("%d - %d", low);  /* woops -- where is the other value to print? */
```

(332) interrupt_level should be 0 to 7 ***(Parser)***

The pragma `interrupt_level` must have an argument from 0 to 7, e.g.:

```
#pragma interrupt_level  /* woops -- what is the level */
void interrupt isr(void)
{
  /* isr code goes here */
}
```

(333) unrecognized qualifier name after "strings" ***(Parser)***

The `pragma strings` was passed a qualifier that was not identified, e.g.:

```
#pragma strings cinst  /* woops -- should that be #pragma strings const ? */
```

(335) unknown pragma * ***(Parser)***

An unknown `pragma` directive was encountered, e.g.:

```
#pragma rugsused w  /* I think you meant regsused */
```

(336) string concatenation across lines ***(Parser)***

Strings on two lines will be concatenated. Check that this is the desired result, e.g.:

```
char * cp = "hi"
  "there";        /* this is okay, but is it what you had intended? */
```

(337) line does not have a newline on the end ***(Parser)***

The last line in the file is missing the *newline* (operating system dependent character) from the end. Some editors will create such files, which can cause problems for include files. The ANSI C standard requires all source files to consist of complete lines only.

(338) can't create * file "*" ***(Code Generator, Assembler, Linker, Optimiser)***

The application tried to create the named file, but it could not be created. Check that all file pathnames are correct.

(338) can't create * file "*" ***(Linker, Code Generator Driver)***

The compiler was unable to create a temporary file. Check the DOS Environment variable TEMP (and TMP) and verify it points to a directory that exists, and that there is space available on that drive. For example, `AUTOEXEC.BAT` should have something like:

```
SET TEMP=C:\TEMP
```

where the directory `C:\TEMP` exists.

(339) initializer in "extern" declaration ***(Parser)***

A declaration containing the keyword `extern` has an initialiser. This overrides the `extern` storage class, since to initialise an object it is necessary to define (i.e. allocate storage for) it, e.g.:

```
extern int other = 99;  /* if it's extern and not allocated storage,
                           how can it be initialized? */
```

(343) implicit return at end of non-void function ***(Parser)***

A function which has been declared to return a value has an execution path that will allow it to reach the end of the function body, thus returning without a value. Either insert a `return` statement with a value, or if the function is not to return a value, declare it `void`, e.g.:

```
int mydiv(double a, int b)
{
  if(b != 0)
    return a/b;    /* what about when b is 0? */
}                  /* warning flagged here */
```

(344) non-void function returns no value ***(Parser)***

A function that is declared as returning a value has a `return` statement that does not specify a return value, e.g.:

```
int get_value(void)
{
  if(flag)
    return val++;
  return;         /* what is the return value in this instance? */
}
```

(345) unreachable code ***(Parser)***

This section of code will never be executed, because there is no execution path by which it could be reached, e.g.:

```
while(1)          /* how does this loop finish? */
  process();
flag = FINISHED;  /* how do we get here? */
```

(346) declaration of * hides outer declaration ***(Parser)***

An object has been declared that has the same name as an outer declaration (i.e. one outside and preceding the current function or block). This is legal, but can lead to accidental use of one variable when the outer one was intended, e.g.:

```
int input;           /* input has filescope */
void process(int a)
{
  int input;         /* local blockscope input */
  a = input;         /* this will use the local variable. Is this right? */
```

(347) external declaration inside function ***(Parser)***

A function contains an `extern` declaration. This is legal but is invariably not desirable as it restricts the scope of the function declaration to the function body. This means that if the compiler encounters another declaration, use or definition of the extern object later in the same file, it will no longer have the earlier declaration and thus will be unable to check that the declarations are consistent. This can lead to strange behaviour of your program or signature errors at link time. It will also hide any previous declarations of the same thing, again subverting the compiler's type checking. As a general rule, always declare `extern` variables and functions outside any other functions. For example:

```
int process(int a)
{
  extern int away;  /* this would be better outside the function */
  return away + a;
}
```

(348) auto variable * should not be qualified ***(Parser)***

An `auto` variable should not have qualifiers such as `near` or `far` associated with it. Its storage class is implicitly defined by the stack organization. An `auto` variable may be qualified with `static`, but it is then no longer `auto`.

(349) non-prototyped function declaration: * ***(Parser)***

A function has been declared using old-style (K&R) arguments. It is preferable to use prototype declarations for all functions, e.g.:

```
int process(input)
int input;         /* warning flagged here */
{
}
```

This would be better written:

```
int process(int input)
{
}
```

(350) unused *: * (from line *) ***(Parser)***

The indicated object was never used in the function or module being compiled. Either this object is redundant, or the code that was meant to use it was excluded from compilation or misspelt the name of the object. Note that the symbols `rcsid` and `sccsid` are never reported as being unused.

(352) float param coerced to double ***(Parser)***

Where a non-prototyped function has a parameter declared as `float`, the compiler converts this into a `double float`. This is because the default C type conversion conventions provide that when a floating point number is passed to a non-prototyped function, it will be converted to `double`. It is important that the function declaration be consistent with this convention, e.g.:

```
double inc_flt(f)  /* the parameter f will be converted to double type */
float f;           /* warning flagged here */
{
  return f * 2;
}
```

(353) sizeof external array "*" is zero ***(Parser)***

The size of an external array evaluates to zero. This is probably due to the array not having an explicit dimension in the extern declaration.

(354) possible pointer truncation ***(Parser)***

A pointer qualified far has been assigned to a default pointer or a pointer qualified near, or a default pointer has been assigned to a pointer qualified near. This may result in truncation of the pointer and loss of information, depending on the memory model in use.

(355) implicit signed to unsigned conversion ***(Parser)***

A `signed` number is being assigned or otherwise converted to a larger `unsigned` type. Under the ANSI "value preserving" rules, this will result in the `signed` value being first sign-extended to a `signed` number the size of the target type, then converted to `unsigned` (which involves no change in bit pattern). Thus an unexpected sign extension can occur. To ensure this does not happen, first convert the signed value to an unsigned equivalent, e.g.:

```
signed char sc;
unsigned int ui;
ui = sc;         /* if sc contains 0xff, ui will contain 0xffff for example */
```

will perform a sign extension of the `char` variable to the longer type. If you do not want this to take place, use a cast, e.g.:

```
ui = (unsigned char)sc;
```

(356) implicit conversion of float to integer ***(Parser)***

A floating point value has been assigned or otherwise converted to an integral type. This could result in truncation of the floating point value. A typecast will make this warning go away.

```
double dd;
int i;
i = dd;    /* is this really what you meant? */
```

If you do intend to use an expression like this, then indicate that this is so by a cast:

```
i = (int)dd;
```

(357) illegal conversion of integer to pointer ***(Parser)***

An integer has been assigned to or otherwise converted to a pointer type. This will usually mean you have used the wrong variable, but if this is genuinely what you want to do, use a typecast to inform the compiler that you want the conversion and the warning will be suppressed. This may also mean you have forgotten the `&` address operator, e.g.:

```
int * ip;
int i;
ip = i;    /* woops -- did you mean ip = &i ? */
```

If you do intend to use an expression like this, then indicate that this is so by a cast:

```
ip = (int *)i;
```

(358) illegal conversion of pointer to integer ***(Parser)***

A pointer has been assigned to or otherwise converted to a integral type. This will usually mean you have used the wrong variable, but if this is genuinely what you want to do, use a typecast to inform the compiler that you want the conversion and the warning will be suppressed. This may also mean you have forgotten the `*` dereference operator, e.g.:

```
int * ip;
int i;
i = ip;    /* woops -- did you mean i = *ip ? */
```

If you do intend to use an expression like this, then indicate that this is so by a cast:

```
i = (int)ip;
```

(359) illegal conversion between pointer types ***(Parser)***

A pointer of one type (i.e. pointing to a particular kind of object) has been converted into a pointer of a different type. This will usually mean you have used the wrong variable, but if this is genuinely what you want to do, use a typecast to inform the compiler that you want the conversion and the warning will be suppressed, e.g.:

```
long input;
char * cp;
cp = &input;  /* is this correct? */
```

This is common way of accessing bytes within a multi-byte variable. To indicate that this is the intended operation of the program, use a cast:

```
cp = (char *)&input;  /* that's better */
```

This warning may also occur when converting between pointers to objects which have the same type, but which have different qualifiers, e.g.:

```
char * cp;
cp = "I am a string of characters";  /* yes, but what sort of characters? */
```

If the default type for string literals is `const char *`, then this warning is quite valid. This should be written:

```
const char * cp;
cp = "I am a string of characters";  /* that's better */
```

Omitting a qualifier from a pointer type is often disastrous, but almost certainly not what you intend.

(360) array index out of bounds ***(Parser)***

An array is being indexed with a constant value that is less than zero, or greater than or equal to the number of elements in the array. This warning will not be issued when accessing an array element via a pointer variable, e.g.:

```
int i, * ip, input[10];
i = input[-2];          /* woops -- this element doesn't exist */
ip = &input[5];
i = ip[-2];             /* this is okay */
```

(361) function declared implicit int ***(Parser)***

Where the compiler encounters a function call of a function whose name is presently undefined, the compiler will automatically declare the function to be of type `int`, with unspecified (K&R style) parameters. If a definition of the function is subsequently encountered, it is possible that its type and arguments will be different from the earlier implicit declaration, causing a compiler error. The solution is to ensure that all functions are defined or at least declared before use, preferably with prototyped parameters. If it is necessary to make a forward declaration of a function, it should be preceded with the keywords `extern` or `static` as appropriate. For example:

```
void set(long a, int b);  /* I may prevent an error arising from calls below */
void main(void)
{
  set(10L, 6);    /* by here a prototype for set should have seen */
}
```

(362) redundant & applied to array ***(Parser)***

The address operator `&` has been applied to an array. Since using the name of an array gives its address anyway, this is unnecessary and has been ignored, e.g.:

```
int array[5];
int * ip;
ip = &array;  /* array is a constant, not a variable; the & is redundant. */
```

(364) attempt to modify * object ***(Parser)***

Objects declared `const` or `code` may not be assigned to or modified in any other way by your program. The effect of attempting to modify such an object is compiler-specific.

```
const int out = 1234;  /* "out" is read only */
out = 0;               /* woops -- writing to a read-only object */
```

(365) pointer to non-static object returned ***(Parser)***

This function returns a pointer to a non-`static` (e.g. `auto`) variable. This is likely to be an error, since the storage associated with automatic variables becomes invalid when the function returns, e.g.:

```
char * get_addr(void)
{
  char c;
  return &c;  /* returning this is dangerous; the pointer could be dereferenced */
}
```

(366) operands of * not same pointer type ***(Parser)***

The operands of this operator are of different pointer types. This probably means you have used the wrong pointer, but if the code is actually what you intended, use a typecast to suppress the error message.

(367) function is already "extern"; can't be "static" ***(Parser)***

This function was already declared `extern`, possibly through an implicit declaration. It has now been redeclared `static`, but this redeclaration is invalid.

```
void main(void)
{
  set(10L, 6);    /* at this point the compiler assumes set is extern... */
}
static void set(long a, int b)  /* now it finds out otherwise */
{
  PORTA = a + b;
}
```

(368) array dimension on *[] ignored ***(Preprocessor)***

An array dimension on a function parameter has been ignored because the argument is actually converted to a pointer when passed. Thus arrays of any size may be passed. Either remove the dimension from the parameter, or define the parameter using pointer syntax, e.g.:

```
int get_first(int array[10])  /* param should be: "int array[]" or "int *" */
{                             /* warning flagged here */
  return array[0];
}
```

(369) signed bitfields not supported *(Parser)*

Only `unsigned` bitfields are supported. If a bitfield is declared to be type `int`, the compiler still treats it as `unsigned`, e.g.:

```
struct {
  signed int sign: 1;    /* this must be unsigned */
  signed int value: 15;
} ;
```

(371) missing basic type: int assumed *(Parser)*

This declaration does not include a basic type, so `int` has been assumed. This declaration is not illegal, but it is preferable to include a basic type to make it clear what is intended, e.g.:

```
char c;
i;      /* don't let the compiler make assumptions, use : int i */
func(); /* ditto, use: extern int func(int); */
```

(372) , expected *(Parser)*

A *comma* was expected here. This could mean you have left out the *comma* between two identifiers in a declaration list. It may also mean that the immediately preceding type name is misspelled, and has thus been interpreted as an identifier, e.g.:

```
unsigned char a;
unsigned chat b; /* thinks: chat & b are unsigned, but where is the comma? */
```

(375) unknown FNREC type * *(Linker)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(376) bad non-zero node in call graph *(Linker)*

The linker has encountered a top level node in the call graph that is referenced from lower down in the call graph. This probably means the program has indirect recursion, which is not allowed when using a compiled stack.

(378) can't create * file "*" *(Hexmate)*

This type of file could not be created. Is the file or a file by this name already in use?

(379) bad record type * ***(Linker)***

This is an internal compiler error. Ensure the object file is a valid HI-TECH object file. Contact HI-TECH Software technical support with details.

(380) unknown record type: * ***(Linker)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(381) record too long (*): * ***(Linker)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(382) incomplete record: type = *, length = * ***(Dump, Xstrip)***

This message is produced by the DUMP or XSTRIP utilities and indicates that the object file is not a valid HI-TECH object file, or that it has been truncated. Contact HI-TECH Support with details.

(383) text record has length too small: * ***(Linker)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(384) assertion failed: file *, line *, expr * ***(Linker, Parser)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(386) can't open error file * ***(Linker)***

The error file specified using the `-e` linker option could not be opened.

(387) illegal or too many -g flags ***(Linker)***

There has been more than one linker `-g` option, or the `-g` option did not have any arguments following. The arguments specify how the segment addresses are calculated.

(388) duplicate -m flag ***(Linker)***

The map file name has been specified to the linker for a second time. This should not occur if you are using a compiler driver. If invoking the linker manually, ensure that only one instance of this option is present on the command line. See Section 13.7.9 for information on the correct syntax for this option.

(389) illegal or too many -o flags ***(Linker)***

This linker `-o` flag is illegal, or another `-o` option has been encountered. A `-o` option to the linker must be immediately followed by a filename with no intervening space.

(390) illegal or too many -p flags ***(Linker)***

There have been too many `-p` options passed to the linker, or a `-p` option was not followed by any arguments. The arguments of separate `-p` options may be combined and separated by *commas*.

(391) missing arg to -Q ***(Linker)***

The `-Q` linker option requires the machine type for an argument.

(392) missing arg to -u ***(Linker)***

The `-U` (undefine) option needs an argument.

(393) missing arg to -w ***(Linker)***

The `-W` option (listing width) needs a numeric argument.

(394) duplicate -d or -h flag ***(Linker)***

The symbol file name has been specified to the linker for a second time. This should not occur if you are using a compiler driver. If invoking the linker manually, ensure that only one instance of either of these options is present on the command line.

(395) missing arg to -j ***(Linker)***

The maximum number of errors before aborting must be specified following the `-j` linker option.

(396) illegal flag -* ***(Linker)***

This linker option is unrecognized.

(398) output file cannot be also an input file ***(Linker)***

The linker has detected an attempt to write its output file over one of its input files. This cannot be done, because it needs to simultaneously read and write input and output files.

(400) bad object code format ***(Linker)***

This is an internal compiler error. The object code format of an object file is invalid. Ensure it is a valid HI-TECH object file. Contact HI-TECH Software technical support with details.

(401) cannot get memory ***(Linker)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(404) bad maximum length value to -<digits> ***(Objtohex)***

The first value to the OBJTOHEX `-n,m` hex length/rounding option is invalid.

(405) bad record size rounding value to -<digits> ***(Objtohex)***

The second value to the OBJTOHEX `-n,m` hex length/rounding option is invalid.

(410) bad combination of flags ***(Objtohex)***

The combination of options supplied to `OBJTOHEX` is invalid.

(412) text does not start at 0 ***(Objtohex)***

Code in some things must start at zero. Here it doesn't.

(413) write error on * ***(Assembler, Linker, Cromwell)***

A write error occurred on the named file. This probably means you have run out of disk space.

(414) read error on * ***(Linker)***

The linker encountered an error trying to read this file.

(415) text offset too low ***(Objtohex)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(416) bad character in extended Tekhex line (*) ***(Objtohex)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(417) seek error ***(Linker)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(418) image too big ***(Objtohex)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(419) object file is not absolute ***(Objtohex)***

The object file passed to `OBJTOHEX` has relocation items in it. This may indicate it is the wrong object file, or that the linker or OBJTOHEX have been given invalid options. The object output files from the assembler are relocatable, not absolute. The object file output of the linker is absolute.

(420) too many relocation items ***(Objtohex)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(421) too many segments ***(Objtohex)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(422) no end record ***(Linker)***

This object file has no end record. This probably means it is not an object file. Contact HI-TECH Support if the object file was generated by the compiler.

(423) illegal record type ***(Linker)***

There is an error in an object file. This is either an invalid object file, or an internal error in the linker. Contact HI-TECH Support with details if the object file was created by the compiler.

(424) record too long ***(Objtohex)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(425) incomplete record ***(Objtohex, Libr)***

The object file passed to OBJTOHEX or the librarian is corrupted. Contact HI-TECH Support with details.

(426) can't open checksum file * ***(Linker)***

The checksum file specified to OBJTOHEX could not be opened. Confirm the spelling and path of the file specified on the command line.

(427) syntax error in checksum list ***(Objtohex)***

There is a syntax error in a checksum list read by OBJTOHEX. The checksum list is read from standard input in response to an option.

(428) too many segment fixups ***(Objtohex)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(429) bad segment fixups ***(Objtohex)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(430) bad checksum specification ***(Objtohex)***

A checksum list supplied to `OBJTOHEX` is syntatically incorrect.

(433) out of memory allocating * blocks of * ***(Linker)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(434) too many symbols (*) ***(Linker)***

There are too many symbols in the symbol table, which has a limit of * symbols. Change some global symbols to local symbols to reduce the number of symbols.

(435) bad segspec * ***(Linker)***

The segment specification option (`-G`) to the linker is invalid, e.g.:

```
-GA/f0+10
```

Did you forget the radix?

```
-GA/f0h+10
```

(436) psect "*" re-orged *(Linker)*

This psect has had its start address specified more than once.

(437) missing "=" in class spec *(Linker)*

A class spec needs an = sign, e.g. -Ctext=ROM See Section 13.7.9 for more infomation.

(438) bad size in -S option *(Linker)*

The address given in a `-S` specification is invalid: it should be a valid number, in decimal, octal or hexadecimal radix. The radix is specified by a trailing `O`, for octal, or `H` for hex. A leading `0x` may also be used for hexadecimal. Case in not important for any number or radix. Decimal is the default, e.g.:

```
-SCODE=f000
```

Did you forget the radix?

```
-SCODE=f000h
```

(441) bad -A spec: "*" *(Linker)*

The format of a `-A` specification, giving address ranges to the linker, is invalid, e.g.:

```
-ACODE
```

What is the range for this class? Maybe you meant:

```
-ACODE=0h-1fffh
```

(443) bad low address in -A spec - * *(Linker)*

The low address given in a -A specification is invalid: it should be a valid number, in decimal, octal or hexadecimal radix. The radix is specified by a trailing O (for octal) or H for hex. A leading `0x` may also be used for hexadecimal. Case in not important for any number or radix. Decimal is default, e.g.:

```
-ACODE=1fff-3fffh
```

Did you forget the radix?

```
-ACODE=1fffh-3fffh
```

(444) expected "-" in -A spec *(Linker)*

There should be a minus sign, `-`, between the high and low addresses in a `-A` linker option, e.g.

```
-AROM=1000h
```

maybe you meant:

```
-AROM=1000h-1fffh
```

(445) bad high address in -A spec - * *(Linker)*

The high address given in a `-A` specification is invalid: it should be a valid number, in decimal, octal or hexadecimal radix. The radix is specified by a trailing `O`, for octal, or `H` for hex. A leading `0x` may also be used for hexadecimal. Case in not important for any number or radix. Decimal is the default, e.g.:

```
-ACODE=0h-ffff
```

Did you forget the radix?

```
-ACODE=0h-ffffh
```

See Section 13.7.20 for more infomation.

(446) bad overrun address in -A spec - * *(Linker)*

The overrun address given in a -A specification is invalid: it should be a valid number, in decimal, octal or hexadecimal radix. The radix is specified by a trailing O (for octal) or H for hex. A leading `0x` may also be used for hexadecimal. Case in not important for any number or radix. Decimal is default, e.g.:

```
-AENTRY=0-0FFh-1FF
```

Did you forget the radix?

```
-AENTRY=0-0FFh-1FFh
```

(447) bad load address in -A spec - * *(Linker)*

The load address given in a `-A` specification is invalid: it should be a valid number, in decimal, octal or hexadecimal radix. The radix is specified by a trailing `O` (for octal) or `H` for hex. A leading `0x` may also be used for hexadecimal. Case in not important for any number or radix. Decimal is default, e.g.:

```
-ACODE=0h-3fffh/a000
```

Did you forget the radix?

```
-ACODE=0h-3fffh/a000h
```

(448) bad repeat count in -A spec - * *(Linker)*

The repeat count given in a `-A` specification is invalid, e.g.:

```
-AENTRY=0-0FFhxf
```

Did you forget the radix?

```
-AENTRY=0-0FFhxfh
```

(449) syntax error in -A spec: * *(Linker)*

The `-A` spec is invalid. A valid -A spec should be something like:
-AROM=1000h-1FFFh

(450) unknown psect: * *(Linker, Optimiser)*

This psect has been listed in a `-P` option, but is not defined in any module within the program.

(451) bad origin format in spec *(Linker)*

The origin format in a `-p` option is not a validly formed decimal, octal or hex number, nor is it the name of an existing psect. A hex number must have a trailing H, e.g.:

```
-pbss=f000
```

Did you forget the radix?

```
-pbss=f000h
```

(452) bad min (+) format in spec ***(Linker)***

The minimum address specification in the linker's `-p` option is badly formatted, e.g.:

```
-pbss=data+f000
```

Did you forget the radix?

```
-pbss=data+f000h
```

(453) missing number after % in -p option ***(Linker)***

The `%` operator in a `-p` option (for rounding boundaries) must have a number after it.

(455) psect * not relocated on 0x* byte boundary ***(Linker)***

This psect is not relocated on the required boundary. Check the relocatability of the psect and correct the `-p` option. if necessary.

(458) cannot open ***(Objtohex)***

`OBJTOHEX` cannot open the specified input file. Confirm the spelling and path of the file specified on the command line.

(462) can't open avmap file * ***(Linker)***

A file required for producing Avocet format symbol files is missing. Confirm the spelling and path of the file specified on the command line.

(463) missing memory key in avmap file ***(Linker)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(464) missing key in avmap file ***(Linker)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(465) undefined symbol in FNBREAK record: * ***(Linker)***

The linker has found an undefined symbol in the `FNBREAK` record for a non-reentrant function. Contact HI-TECH Support if this is not handwritten assembler code.

(466) undefined symbol in FNINDIR record: * ***(Linker)***

The linker has found an undefined symbol in the `FNINDIR` record for a non-reentrant function. Contact HI-TECH Support if this is not handwritten assembler code.

(467) undefined symbol in FNADDR record: * ***(Linker)***

The linker has found an undefined symbol in the FNADDR record for a non-reentrant function. Contact HI-TECH Support if this is not handwritten assembler code.

(468) undefined symbol in FNCALL record: * ***(Linker)***

The linker has found an undefined symbol in the `FNCALL` record for a non-reentrant function. Contact HI-TECH Support if this is not handwritten assembler code.

(469) undefined symbol in FNROOT record: * ***(Linker)***

The linker has found an undefined symbol in the `FNROOT` record for a non-reentrant function. Contact HI-TECH Support if this is not handwritten assembler code.

(470) undefined symbol in FNSIZE record: * ***(Linker)***

The linker has found an undefined symbol in the `FNSIZE` record for a non-reentrant function. Contact HI-TECH Support if this is not handwritten assembler code.

(471) recursive function calls: ***(Linker)***

These functions (or function) call each other recursively. One or more of these functions has statically allocated local variables (compiled stack). Either use the `reentrant` keyword (if supported with this compiler) or recode to avoid recursion, e.g.:

```
int test(int a)
{
  if(a == 5)
    return test(a++);  /* recursion may not be supported by some compilers */
  return 0;
}
```

(472) function * appears in multiple call graphs: rooted at * and * ***(Linker)***

This function can be called from both main-line code and interrupt code. Use the `reentrant` keyword, if this compiler supports it, or recode to avoid using local variables or parameters, or duplicate the function, e.g.:

```
void interrupt my_isr(void)
{
  scan(6);              /* scan is called from an interrupt function */
}
void process(int a)
{
  scan(a);              /* scan is also called from main-line code */
}
```

(474) no psect specified for function variable/argument allocation ***(Linker)***

The `FNCONF` assembler directive which specifies to the linker information regarding the auto/parameter block was never seen. This is supplied in the standard runtime files if necessary. This error may imply that the correct run-time startoff module was not linked. Ensure you have used the `FNCONF` directive if the runtime startup module is hand-written.

(475) conflicting FNCONF records ***(Linker)***

The linker has seen two conflicting `FNCONF` directives. This directive should only be specified once and is included in the standard runtime startup code which is normally linked into every program.

(476) fixup overflow referencing * * (loc 0x* (0x*+*), size *, value 0x*) ***(Linker)***

The linker was asked to relocate (fixup) an item that would not fit back into the space after relocation. See the following error message (477) for more information..

(477) fixup overflow in expression (loc 0x* (0x*+*), size *, value 0x*) ***(Linker)***

Fixup is the process conducted by the linker of replacing symbolic references to variables etc, in an assembler instruction with an absolute value. This takes place after positioning the psects (program sections or blocks) into the available memory on the target device. Fixup overflow is when the value determined for a symbol is too large to fit within the allocated space within the assembler instruction. For example, if an assembler instruction has an 8-bit field to hold an address and the linker determines that the symbol that has been used to represent this address has the value 0x110, then clearly this value cannot be inserted into the instruction.

The causes for this can be many, but hand-written assembler code is always the first suspect. Badly written C code can also generate assembler that ultimately generates fixup overflow errors. Consider the following error message.

```
main.obj: 8: Fixup overflow in expression (loc 0x1FD (0x1FC+1), size 1, value 0x7FC)
```

This indicates that the file causing the problem was `main.obj`. This would be typically be the output of compiling `main.c` or `main.as`. This tells you the file in which you should be looking. The next number (8 in this example) is the record number in the object file that was causing the problem. If you use the `DUMP` utility to examine the object file, you can identify the record, however you do not normally need to do this.

The location (`loc`) of the instruction (0x1FD), the `size` (in bytes) of the field in the instruction for the value (1) , and the `value` which is the actual value the symbol represents, is typically the only information needed to track down the cause of this error. Note that a size which is not a multiple of 8 bits will be rounded up to the nearest byte size, i.e. a 7 bit space in an instruction will be shown as 1 byte.

Generate an assembler list file for the appropriate module. Look for the address specified in the error message.

```
7    07FC    0E21  movlw 33
8    07FD    6FFC  movwf _foo
9    07FE    0012  return
```

and to confirm, look for the symbol referenced in the assembler instruction at this address in the symbol table at the bottom of the same file.

```
Symbol Table                    Fri Aug 12 13:17:37 2004
_foo 01FC   _main 07FF
```

In this example, the instruction causing the problem takes an 8-bit offset into a bank of memory, but clearly the address 0x1FC exceeds this size. Maybe the instruction should have been written as:

```
movwf  (_foo&0ffh)
```

which masks out the top bits of the address containing the bank information.

If the assembler instruction that caused this error was generated by the compiler, in the assembler list file look back up the file from the instruction at fault to determine which C statement has generated this instruction. You will then need to examine the C code for possible errors. incorrectly qualified pointers are an common trigger.

(479) circular indirect definition of symbol * *(Linker)*

The specified symbol has been equated to an external symbol which, in turn, has been equated to the first symbol.

(480) signatures do not match: * (*): 0x*/0x* *(Linker)*

The specified function has different signatures in different modules. This means it has been declared differently, e.g. it may have been prototyped in one module and not another. Check what declarations for the function are visible in the two modules specified and make sure they are compatible, e.g.:

```
extern int get_value(int in);
/* and in another module: */
int get_value(int in, char type)  /* this is different to the declaration */
{
```

(481) common symbol psect conflict: * *(Linker)*

A common symbol has been defined to be in more than one psect.

(482) symbol "*" multiply defined in file "*" *(Assembler)*

This symbol has been defined in more than one place. The assembler will issue this error if a symbol is defined more than once in the same module, e.g.:

```
_next:
  move r0, #55
  move [r1], r0
_next:           ; woops -- choose a different name
```

The linker will issue this warning if the symbol (C or assembler) was defined multiple times in different modules. The names of the modules are given in the error message. Note that C identifiers often have an *underscore* prepended to their name after compilation.

(483) symbol * cannot be global *(Linker)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(484) psect * cannot be in classes * and * ***(Linker)***

A psect cannot be in more than one class. This is either due to assembler modules with conflicting `class=` options to the PSECT directive, or use of the `-C` option to the linker, e.g.:

```
psect final,class=CODE
finish:
/* elsewhere: */
psect final,class=ENTRY
```

(485) unknown "with" psect referenced by psect * ***(Linker)***

The specified psect has been placed with a psect using the psect `with` flag. The psect it has been placed with does not exist, e.g.:

```
psect starttext,class=CODE,with=rext ; was that meant to be with text?
```

(486) psect * selector value redefined ***(Linker)***

The selector associated with this psect has been defined differently in two or more places.

(486) psect * selector value redefined ***(Linker)***

The selector value for this psect has been defined more than once.

(487) psect * type redefined: */* ***(Linker)***

This psect has had its type defined differently by different modules. This probably means you are trying to link incompatible object modules, e.g. linking 386 flat model code with 8086 real mode code.

(488) psect * memory space redefined: */* ***(Linker)***

A global psect has been defined in two different memory spaces. Either rename one of the psects or, if they are the same psect, place them in the same memory space using the `space` psect flag, e.g.:

```
psect spdata,class=RAM,space=0
  ds 6
; elsewhere:
psect spdata,class=RAM,space=1
```

(489) psect * memory delta redefined: */* *(Linker)*

A global psect has been defined with two different delta values, e.g.:

```
psect final,class=CODE,delta=2
finish:
; elsewhere:
psect final,class=CODE,delta=1
```

(490) class * memory space redefined: */* *(Linker)*

A class has been defined in two different memory spaces. Either rename one of the classes or, if they are the same class, place them in the same memory space.

(491) can't find * words for psect "*" in segment "*" *(Linker)*

One of the main tasks the linker performs is positioning the blocks (or psects) of code and data that is generated from the program into the memory available for the target device. This error indicates that the linker was unable to find an area of free memory large enough to accomodate one of the psects. The error message indicates the name of the psect that the linker was attempting to position and the segment name which is typically the name of a class which is defined with a linker `-A` option.

Section **??** lists each compiler-generated psect and what it contains. Typically psect names which are, or include, `text` relate to program code. Names such as `bss` or `data` refer to variable blocks. This error can be due to two reasons.

First, the size of the program or the progam's data has exceeded the total amount of space on the selected device. In other words, some part of your device's memory has completely filled. If this is the case, then the size of the specified psect must be reduced.

The second cause of this message is when the total amount of memory needed by the psect being positioned is sufficient, but that this memory is fragmented in such a way that the largest contiguous block is too small to accomodate the psect. The linker is unable to split psects in this situation. That is, the linker cannot place part of a psect at one location and part somewhere else. Thus, the linker must be able to find a contiguous block of memory large enough for every psect. If this is the cause of the error, then the psect must be split into smaller psects if possible.

To find out what memory is still available, generate and look in the map file, see Section 10.4.9 for information on how to generate a map file. Search for the string `UNUSED ADDRESS RANGES`. Under this heading, look for the name of the segment specified in the error message. If the name is not present, then all the memory available for this psect has been allocated. If it is present, there will be one address range specified under this segment for each free block of memory. Determine the size of each block and compare this with the number of words specified in the error message.

Psects containing code can be reduced by using all the compiler's optimizations, or restructring the program. If a code psect must be split into two or more small psects, this requies splitting a function into two or more smaller functions (which may call each other). These functions may need to be placed in new modules.

Psects containing data may be reduced when invoking the compiler optimizations, but the effect is less dramatic. The program may need to be rewritten so that it needs less variables. Section 13.9.1 has information on interpreting the map file's call graph if the compiler you are using uses a compiled stack. (If the string `Call grpah:` is not present in the map file, then the compiled code uses a hardware stack.) If a data psect needs to be split into smaller psects, the definitions for variables will need to be moved to new modules or more evenly spread in the existing modules. Memory allocation for `auto` variables is entirely handled by the compiler. Other than reducing the number of these variables used, the programmer has little control over their operation. This applies whether the compiled code uses a hardware or compiled stack.

For example, after receiving the message:

```
Can't find 0x34 words (0x34 withtotal) for psect text in segment CODE (error)
```

look in the map file for the ranges of unused memory.

```
UNUSED ADDRESS RANGES
        CODE                00000244-0000025F
                            00001000-0000102f
        RAM                 00300014-00301FFB
```

In the `CODE` segment, there is 0x1c (0x25f-0x244+1) bytes of space available in one block and 0x30 available in another block. Neither of these are large enough to accomodate the psect `text` which is 0x34 bytes long. Notice, however, that the total amout of memory available is larger than 0x34 bytes.

(492) psect is absolute: * ***(Linker)***

This psect is absolute and should not have an address specified in a `-P` option. Either remove the `abs` psect flag, or remove the `-P` linker option.

(493) psect origin multiply defined: * ***(Linker)***

The origin of this psect is defined more than once. There is most likely more than one `-p` linker option specifying this psect.

(494) bad -P format "*"/"*" ***(Linker)***

The `-P` option given to the linker is malformed. This option specifies placement of a psect, e.g.:

```
-Ptext=10g0h
```

Maybe you meant:

```
-Ptext=10f0h
```

(497) psect exceeds max size: *: *h > *h ***(Linker)***

The psect has more bytes in it than the maximum allowed as specified using the `size` psect flag.

(498) psect exceeds address limit: *: *h > *h ***(Linker)***

The maximum address of the psect exceeds the limit placed on it using the `limit` psect flag. Either the psect needs to be linked at a different location or there is too much code/data in the psect.

(499) undefined symbol: ***(Assembler, Linker)***

The symbol following is undefined at link time. This could be due to spelling error, or failure to link an appropriate module.

(500) undefined symbols: ***(Linker)***

A list of symbols follows that were undefined at link time. These errors could be due to spelling error, or failure to link an appropriate module.

(501) entry point multiply defined ***(Linker)***

There is more than one entry point defined in the object files given the linker. End entry point is specified after the `END` directive. The runtime startup code defines the entry point, e.g.:

```
powerup:
  goto start
  END  powerup  ; end of file and define entry point
; other files that use END should not define another entry point
```

(502) incomplete * record body: length = * ***(Linker)***

An object file contained a record with an illegal size. This probably means the file is truncated or not an object file. Contact HI-TECH Support with details.

(503) ident records do not match ***(Linker)***

The object files passed to the linker do not have matching ident records. This means they are for different processor types.

(504) object code version is greater than *.* ***(Linker)***

The object code version of an object module is higher than the highest version the linker is known to work with. Check that you are using the correct linker. Contact HI-TECH Support if the object file if you have not patched the linker.

(505) no end record found ***(Linker)***

An object file did not contain an end record. This probably means the file is corrupted or not an object file. Contact HI-TECH Support if the object file was generated by the compiler.

(506) record too long: *+* ***(Linker)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(507) unexpected end of file ***(Linker)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(508) relocation offset * out of range 0..*-*-1 ***(Linker)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(509) illegal relocation size: * ***(Linker)***

There is an error in the object code format read by the linker. This either means you are using a linker that is out of date, or that there is an internal error in the assembler or linker. Contact HI-TECH Support with details if the object file was created by the compiler.

(510) complex relocation not supported for -r or -l options ***(Linker)***

The linker was given a `-R` or `-L` option with file that contain complex relocation.

(511) bad complex range check ***(Linker)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(512) unknown complex operator 0x* ***(Linker)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(513) bad complex relocation ***(Linker)***

The linker has been asked to perform complex relocation that is not syntactically correct. Probably means an object file is corrupted.

(514) illegal relocation type: * ***(Linker)***

An object file contained a relocation record with an illegal relocation type. This probably means the file is corrupted or not an object file. Contact HI-TECH Support with details if the object file was created by the compiler.

(515) unknown symbol type * ***(Linker)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(516) text record has bad length: *-*-(*+1) < 0 ***(Linker)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(517) write error (out of disk space?) * ***(Linker)***

A write error occurred on the named file. This probably means you have run out of disk space.

(519) can't seek in * ***(Linker)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(520) function * is never called ***(Linker)***

This function is never called. This may not represent a problem, but space could be saved by removing it. If you believe this function should be called, check your source code. Some assembler library routines are never called, although they are actually execute. In this case, the routines are linked in a special sequence so that program execution falls through from one routine to the next.

(521) call depth exceeded by * ***(Linker)***

The call graph shows that functions are nested to a depth greater than specified.

(522) library * is badly ordered ***(Linker)***

This library is badly ordered. It will still link correctly, but it will link faster if better ordered.

(523) argument -W* ignored ***(Linker)***

The argument to the linker option `-w` is out of range. This option controls two features. For warning levels, the range is -9 to 9. For the map file width, the range is greater than or equal to 10.

(524) unable to open list file * ***(Linker)***

The named list file could not be opened. The linker would be trying to fixup the list file so that it will contain absolute addresses. Ensure that an assembler list file was generated during the compilation stage. Alternatively, remove the assembler list file generation option from the link step.

(525) too many address spaces - space * ignored ***(Linker)***

The limit to the number of address spaces (specified with the `PSECT` assembler directive) is currently 16.

(526) psect * not specified in -p option (first appears in *) ***(Linker)***

This psect was not specified in a `-P` or `-A` option to the linker. It has been linked at the end of the program, which is probably not where you wanted it.

(528) no start record: entry point defaults to zero ***(Linker)***

None of the object files passed to the linker contained a start record. The start address of the program has been set to zero. This may be harmless, but it is recommended that you define a start address in your startup module by using the `END` directive.

(593) can't find 0x* words (0x* withtotal) for psect * in segment * ***(Linker)***

See error (491) in Appendix B.

(596) segment *(*-*) overlaps segment *(*-*) ***(Linker)***

The named segments have overlapping code or data. Check the addresses being assigned by the `-P` linker option.

(597) can't open ***(Linker)***

An object file could not be opened. Confirm the spelling and path of the file specified on the command line.

(602) null format name ***(Cromwell)***

The `-I` or `-O` option to Cromwell must specify a file format.

(603) ambiguous format name "*" ***(Cromwell)***

The input or output format specified to Cromwell is ambiguous. These formats are specified with the `-i`*`key`* and `-o`*`key`* options respectively.

(604) unknown format name "*" ***(Cromwell)***

The output format specified to `CROMWELL` is unknown, e.g.:

```
cromwell -m -P16F877 main.hex main.sym -ocot
```

and output file type of `cot`, did you mean `cof`?

(605) did not recognize format of input file ***(Cromwell)***

The input file to Cromwell is required to be COD, Intel HEX, Motorola HEX, COFF, OMF51, P&E or HI-TECH.

(606) inconsistent symbol tables ***(Cromwell)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(607) inconsistent line number tables ***(Cromwell)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(609) missing processor spec after -P ***(Cromwell)***

The `-p` option to cromwell must specify a processor name.

(611) too many input files ***(Cromwell)***

To many input files have been specified to be converted by `CROMWELL`.

(612) too many output files ***(Cromwell)***

To many output file formats have been specified to `CROMWELL`.

(613) no output file format specified ***(Cromwell)***

The output format must be specified to CROMWELL.

(614) no input files specified ***(Cromwell)***

CROMWELL must have an input file to convert.

(619) I/O error reading symbol table

Cromwell could not read the symbol table. This could be because the file was truncated or there was some other problem reading the file. Contact HI-TECH Support with details.

(620) file name index out of range in line number record ***(Cromwell)***

The COD file has an invalid format in the specified record.

(625) too many files in COFF file ***(Cromwell)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(626) string lookup failed in coff:get_string() ***(Cromwell)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(634) error dumping * ***(Cromwell)***

Either the input file to `CROMWELL` is of an unsupported type or that file cannot be dumped to the screen.

(635) invalid hex file: *, line * ***(Cromwell)***

The specified HEX file contains an invalid line. Contact HI-TECH Support if the HEX file was generated by the compiler.

(636) checksum error in Intel hex file *, line * *(Cromwell, Hexmate)*

A checksum error was found at the specified line in the specified Intel hex file. The HEX file may be corrupt.

(674) too many references to * *(Cref)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(675) can't open * for input *(Cref)*

CREF cannot open the specified input file. Confirm the spelling and path of the file specified on the command line.

(676) can't open * for output *(Cref)*

CREF cannot open the specified output file. Confirm the spelling and path of the file specified on the command line.

(679) unknown extraspecial: * *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(680) bad format for -P option *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(685) bad putwsize *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(686) bad switch size * *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(687) bad pushreg "*" *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details. See Section 13.7.2 for more infomation.

(688) bad popreg "*" ***(Code Generator)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(689) unknown predicate * ***(Code Generator)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(692) interrupt function "*" may only have one interrupt level ***(Code Generator)***

Only one interrupt level may be associated with an `interrupt` function. Check to ensure that only one `interrupt_level` pragma has been used with the function specified. This pragma may be used more than once on main-line functions that are called from `interrupt` functions. For example:

```
#pragma interrupt_level 0
#pragma interrupt_level 1  /* which is it to be: 0 or 1? */
void interrupt isr(void)
{
```

(693) interrupt level may only be 0 (default) or 1 ***(Code Generator)***

The only possible interrupt levels are 0 or 1. Check to ensure that all `interrupt_level` pragmas use these levels.

```
#pragma interrupt_level 2  /* woops -- only 0 or 1 */
void interrupt isr(void)
{
  /* isr code goes here */
}
```

(695) duplicate case label * ***(Code Generator)***

There are two case labels with the same value in this `switch` statement, e.g.:

```
switch(in) {
case '0':  /* if this is case '0'... */
  b++;
  break;
case '0':  /* then what is this case? */
  b--;
  break;
}
```

(696) out-of-range case label * *(Code Generator)*

This case label is not a value that the controlling expression can yield, and thus this label will never be selected.

(697) non-constant case label *(Code Generator)*

A `case` label in this `switch` statement has a value which is not a constant.

(699) no case labels *(Code Generator)*

There are no `case` labels in this `switch` statement, e.g.:

```
switch(input) {
}               /* there is nothing to match the value of input */
```

(701) unreasonable matching depth *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(702) regused - bad arg to G *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(703) bad GN *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details. See Section 13.7.2 for more infomation.

(704) bad RET_MASK *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(705) bad which (*) after I *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(706) expand - bad which *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(707) bad SX *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.See Section 13.7.20 for more infomation.

(708) bad mod "+" for how = * *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(709) metaregister * can't be used directly *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(710) bad U usage *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(711) expand - bad how *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(712) can't generate code for this expression *(Code Generator)*

This error indicates that a C expression is too difficult for the code generator to actually compile. For successful code generation, the code generator must know how to compile an expression and there must be enough resources (e.g. registers or temporary memory locations) available. Simplifying the expression, e.g. using a temporary variable to hold an intermediate result, may get around this message. Contact HI-TECH Support with details of this message.

This error may also be issued if the code being compiled is in some way unusual. For example code which writes to a const-qualified object is illegal and will result in warning messages, but the code generator may unsuccessfully try to produce code to perform the write.

(714) bad intermediate code *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(715) bad pragma * *(Code Generator)*

The code generator has been passed a `pragma` directive that it does not understand. This implies that the pragma you have used is a HI-TECH specific pragma, but the specific compiler you are using has not implemented this pragma.

(716) bad -M option: -M* *(Code Generator)*

The code generator has been passed a `-M` option that it does not understand. This should not happen if it is being invoked by a standard compiler driver.

(717) illegal switch * *(Code Generator, Assembler, Optimiser)*

This command line option was not understood.

(718) incompatible intermediate code version; should be *.* *(Code Generator)*

The intermediate code file produced by P1 is not the correct version for use with this code generator. This is either that incompatible versions of one or more compilers have been installed in the same directory, or a temporary file error has occurred leading to corruption of a temporary file. Check the setting of the TEMP environment variable. If it refers to a long path name, change it to something shorter. Contact HI-TECH Support with details if required.

(720) multiple free: * *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(721) bad element count expr *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(722) bad variable syntax *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(723) functions nested too deep *(Code Generator)*

This error is unlikely to happen with C code, since C cannot have nested functions! Contact HI-TECH Support with details.

(724) bad op * to revlog *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(726) bad uconval - * *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(727) bad bconfloat - * ***(Code Generator)***

This is an internal code generator error. Contact HI-TECH technical support with details.

(728) bad confloat - * ***(Code Generator)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(729) bad conval - * ***(Code Generator)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(730) bad op: "*" ***(Code Generator)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(731) expression error with reserved word ***(Code Generator)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(732) can't initialize bit type ***(Code Generator)***

Variables of type `bit` cannot be initialised, e.g.:

```
bit b1 = 1;  /* woops -- b1 must be assigned a value after its definition */
```

(733) bad string "*" in psect pragma ***(Code Generator)***

The code generator has been passed a `pragma psect` directive that has a badly formed string, e.g.:

```
#pragma psect text  /* redirect text psect into what? */
```

Maybe you meant something like:

```
#pragma psect text=special_text
```

(734) too many psect pragmas ***(Code Generator)***

Too many `#pragma psect` directives have been used.

(737) unknown argument to "pragma switch": * ***(Code Generator)***

The `#pragma switch` directive has been used with an invalid switch code generation method. Possible arguments are: `auto`, `simple` and `direct`.

(739) error closing output file ***(Code Generator, Optimiser)***

The compiler detected an error when closing a file. Contact HI-TECH Support with details.

(740) bad dimensions ***(Code Generator)***

The code generator has been passed a declaration that results in an array having a zero dimension.

(741) bit field too large (* bits) ***(Code Generator)***

The maximum number of bits in a bit field is the same as the number of bits in an `int`, e.g. assuming an `int` is 16 bits wide:

```
struct {
  unsigned flag  : 1;
  unsigned value : 12;
  unsigned cont  : 6;   /* woops -- that makes a total of 19 bits */
} object;
```

(742) function "*" argument evaluation overlapped ***(Linker)***

A function call involves arguments which overlap between two functions. This could occur with a call like:

```
void fn1(void)
{
  fn3( 7, fn2(3), fn2(9));    /* Offending call */
}
char fn2(char fred)
{
  return fred + fn3(5,1,0);
}
char fn3(char one, char two, char three)
{
  return one+two+three;
}
```

where fn1 is calling fn3, and two arguments are evaluated by calling fn2, which in turn calls fn3. The program structure should be modified to prevent this type of call sequence.

(744) static object has zero size: * *(Code Generator)*

A static object has been declared, but has a size of zero.

(745) nodecount = * *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(747) unrecognized option to -Z: * *(Code Generator)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(748) variable may be used before set: * *(Code Generator)*

This variable may be used before it has been assigned a value. Since it is an auto variable, this will result in it having a random value, e.g.:

```
void main(void)
{
  int a;
  if(a)         /* woops -- a has never been assigned a value */
    process();
}
```

(749) unknown register name * *(Linker)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(750) constant operand to || or && *(Code Generator)*

One operand to the logical operators || or && is a constant. Check the expression for missing or badly placed parentheses. This message may also occur if the global optimizer is enabled and one of the operands is an auto or static local variable whose value has been tracked by the code generator, e.g.:

```
{
int a;
a = 6;
```

```
if(a || b)  /* a is 6, therefore this is always true */
    b++;
```

(751) arithmetic overflow in constant expression ***(Code Generator)***

A constant expression has been evaluated by the code generator that has resulted in a value that is too big for the type of the expression. The most common code to trigger this warning is assignments to `signed` data types. For example:

```
signed char c;
c = 0xFF;
```

As a `signed` 8-bit quantity, `c` can only be assigned values -128 to 127. The constant is equal to 255 and is outside this range. If you mean to set all bits in this variable, then use either of:

```
c = ~0x0;
c = -1;
```

which will set all the bits in the variable regardless of the size of the variable and without warning.

This warning can also be triggered by intermediate values overflowing. For example:

```
unsigned int i;  /* assume ints are 16 bits wide */
i = 240 * 137;   /* this should be okay, right? */
```

A quick check with your calculator reveils that 240 * 137 is 32880 which can easily be stored in an `unsigned int`, but a warning is produced. Why? Because 240 and 137 and both `signed int` values. Therefore the result of the multiplication must also be a `signed int` value, but a `signed int` cannot hold the value 32880. (Both operands are constant values so the code generator can evaluate this expression at compile time, but it must do so following all the ANSI rules.) The following code forces the multiplication to be performed with an `unsigned` result:

```
i = 240u * 137;  /* force at least one operand to be unsigned */
```

(752) conversion to shorter data type ***(Code Generator)***

Truncation may occur in this expression as the lvalue is of shorter type than the rvalue, e.g.:

```
char a;
int b, c;
a = b + c;  /* conversion of int to char may result in truncation */
```

(753) undefined shift (* bits) ***(Code Generator)***

An attempt has been made to shift a value by a number of bits equal to or greater than the number of bits in the data type. This will produce an undefined result on many processors. This is non-portable code and is flagged as having undefined results by the C Standard, e.g.:

```
int input;
input <<= 33;  /* woops -- that shifts the entire value out of input */
```

(754) bitfield comparison out of range ***(Code Generator)***

This is the result of comparing a bitfield with a value when the value is out of range of the bitfield. For example, comparing a 2-bit bitfield to the value 5 will never be true as a 2-bit bitfield has a range from 0 to 3, e.g.:

```
struct {
  unsigned mask : 2;  /* mask can hold values 0 to 3 */
} value;
int compare(void)
{
  return (value.mask == 6);  /* test can
}
```

(755) division by zero ***(Code Generator)***

A constant expression that was being evaluated involved a division by zero, e.g.:

```
a /= 0;  /* divide by 0: was this what you were intending */
```

(757) constant conditional branch ***(Code Generator)***

A conditional branch (generated by an `if`, `for`, `while` statement etc.) always follows the same path. This will be some sort of comparison involving a variable and a constant expression. For the code generator to issue this message, the variable must have local scope (either `auto` or `static` local) and the global optimizer must be enabled, possibly at higher level than 1, and the warning level threshold may need to be lower than the default level of 0.

The global optimizer keeps track of the contents of local variables for as long as is possible during a function. For C code that compares these variables to constants, the result of the comparison can be deduced at compile time and the output code hard coded to avoid the comparison, e.g.:

```
{
  int a, b;
  a = 5;
  if(a == 4)  /* this can never be false; always perform the true statement */
    b = 6;
```

will produce code that sets `a` to 5, then immediately sets `b` to 6. No code will be produced for the comparison `if(a == 4)`. If a was a global variable, it may be that other functions (particularly interrupt functions) may modify it and so tracking the variable cannot be performed.

This warning may indicate more than an optimization made by the compiler. It may indicate an expression with missing or badly placed parentheses, causing the evaluation to yield a value different to what you expected.

This warning may also be issued because you have written something like `while(1)`. To produce an infinite loop, use `for(;;)`.

A similar situation arises with for loops, e.g.:

```
{
  int a, b;
  for(a=0; a!=10; a++)  /* this loop must iterate at least once */
    b = func(a);
```

In this case the code generator can again pick up that `a` is assigned the value 0, then immediately checked to see if it is equal to 10. Because `a` is modified during the `for` loop, the comparison code cannot be removed, but the code generator will adjust the code so that the comparison is not performed on the first pass of the loop; only on the subsequent passes. This may not reduce code size, but it will speed program execution.

(758) constant conditional branch: possible use of = instead of == ***(Code Generator)***

There is an expression inside an `if` or other conditional construct, where a constant is being assigned to a variable. This may mean you have inadvertently used an assignment = instead of a compare ==, e.g.:

```
  int a, b;
  if(a = 4)  /* this can never be false; always perform the true statement */
    b = 6;
```

will assign the value 4 to a, then , as the value of the assignment is always true, the comparison can be omitted and the assignment to `b` always made. Did you mean:

```
if(a == 4)  /* this can never be false; always perform the true statement */
    b = 6;
```

which checks to see if a is equal to 4.

(759) expression generates no code ***(Code Generator)***

This expression generates no output code. Check for things like leaving off the parentheses in a function call, e.g.:

```
int fred;
fred;     /* this is valid, but has no effect at all */
```

Some devices require that special function register need to be read to clear hardware flags. To accommodate this, in some instances the code generator *does* produce code for a statement which only consists of a variable ID. This may happen for variables which are qualified as `volatile`. Typically the output code will read the variable, but not do anything with the value read.

(760) portion of expression has no effect ***(Code Generator)***

Part of this expression has no side effects, and no effect on the value of the expression, e.g.:

```
int a, b, c;
a = b,c;  /* "b" has no effect, was that meant to be a comma? */
```

(761) sizeof yields 0 ***(Code Generator)***

The code generator has taken the size of an object and found it to be zero. This almost certainly indicates an error in your declaration of a pointer, e.g. you may have declared a pointer to a zero length array. In general, pointers to arrays are of little use. If you require a pointer to an array of objects of unknown length, you only need a pointer to a single object that can then be indexed or incremented.

(763) constant left operand to ? ***(Code Generator)***

The left operand to a conditional operator `?` is constant, thus the result of the tertiary operator `?:` will always be the same, e.g.:

```
a = 8 ? b : c;  /* this is the same as saying a = b; */
```

(764) mismatched comparison *(Code Generator)*

A comparison is being made between a variable or expression and a constant value which is not in the range of possible values for that expression, e.g.:

```
unsigned char c;
if(c > 300)      /* woops -- how can this be true? */
  close();
```

(765) degenerate unsigned comparison *(Code Generator)*

There is a comparison of an `unsigned` value with zero, which will always be true or false, e.g.:

```
unsigned char c;
if(c >= 0)
```

will always be true, because an `unsigned` value can never be less than zero.

(766) degenerate signed comparison *(Code Generator)*

There is a comparison of a `signed` value with the most negative value possible for this type, such that the comparison will always be true or false, e.g.:

```
char c;
if(c >= -128)
```

will always be true, because an 8 bit signed char has a maximum negative value of -128.

(768) constant relational expression *(Code Generator)*

There is a relational expression that will always be true or false. This may be because e.g. you are comparing an `unsigned` number with a negative value, or comparing a variable with a value greater than the largest number it can represent, e.g.:

```
unsigned int a;
if(a == -10)    /* if a is unsigned, how can it be -10? */
  b = 9;
```

(769) no space for macro definition *(Assembler)*

The assembler has run out of memory.

(770) insufficient memory for macro definition ***(Assembler)***

There is not sufficient memory to store a macro definition.

(772) include files nested too deep ***(Assembler)***

Macro expansions and include file handling have filled up the assembler's internal stack. The maximum number of open macros and include files is 30.

(773) macro expansions nested too deep ***(Assembler)***

Macro expansions in the assembler are nested too deep. The limit is 30 macros and include files nested at one time.

(774) too many macro parameters ***(Assembler)***

There are too many macro parameters on this macro definition.

(778) write error on object file ***(Assembler)***

An error was reported when the assembler was attempting to write an object file. This probably means there is not enough disk space.

(779) bad relocation type 0x* ***(Assembler)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(780) too many psects ***(Assembler)***

There are too many psects defined! Boy, what a program!

(781) can't enter abs psect ***(Assembler)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(782) REMSYM error ***(Assembler)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(783) "with=" flags are cyclic *(Assembler)*

If Psect A is to be placed "with" Psect B, and Psect B is to be placed "with" Psect A, there is no hierarchy. The `with` flag is an attribute of a psect and indicates that this psect must be placed in the same memory page as the specified psect.

Remove a `with` flag from one of the psect declarations. Such an assembler declaration may look like:

```
psect my_text,local,class=CODE,with=basecode
```

which will define a psect called `my_text` and place this in the same page as the psect `basecode`.

(784) overfreed *(Assembler)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(785) too many temporary labels *(Assembler)*

There are too many temporary labels in this assembler file. The assembler allows a maximum of 2000 temporary labels.

(787) copyexpr: can't handle v_rtype = * *(Assembler)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(788) invalid character ("*") in number *(Assembler)*

A number contained a character that was not part of the range 0-9 or 0-F.

(790) EOF inside conditional *(Assembler)*

END-of-FILE was encountered while scanning for an "endif" to match a previous "if".

(791) EOF inside macro definition *(Assembler)*

End-of-file was encountered while processing a macro definition. This means there is a missing `ENDM` directive, e.g.:

```
unterm MACRO
  mov  r0, #55
  mov  [r1], r0  ; where is the ENDM?
; end of file
```

(793) unterminated macro arg ***(Assembler)***

An argument to a macro is not terminated. Note that angle brackets ("< >") are used to quote macro arguments.

(794) invalid number syntax ***(Assembler, Optimiser)***

The syntax of a number is invalid. This can be, e.g. use of 8 or 9 in an octal number, or other malformed numbers.

(796) local illegal outside macros ***(Assembler)***

The `LOCAL` directive is only legal inside macros. It defines local labels that will be unique for each invocation of the macro.

(798) macro argument may not appear after LOCAL ***(Assembler)***

The list of labels after the directive `LOCAL` may not include any of the formal parameters to the macro, e.g.:

```
mmm macro a1
  move  r0, #a1
  LOCAL a1       ; woops -- the macro parameter cannot be used with local
ENDM
```

(799) rept argument must be >= 0 ***(Assembler)***

The argument to a REPT directive must be greater than zero, e.g.:

```
rept -2            ; -2 copies of this code? */
  move r0, [r1]++
endm
```

(800) undefined symbol * ***(Assembler)***

The named symbol is not defined in this module, and has not been specified `GLOBAL`.

(801) range check too complex ***(Assembler)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(802) invalid address after "end" directive ***(Assembler)***

The start address of the program which is specified after the assembler `END` directive must be a label in the current file.

(803) undefined temporary label ***(Assembler)***

A temporary label has been referenced that is not defined. Note that a temporary label must have a number >= 0.

(808) add_reloc - bad size ***(Assembler)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(809) unknown addressing mode * ***(Assembler, Optimiser)***

An unknown addressing mode was used in the assembly file.

(810) unknown op in emasm(): * ***(Assembler)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(812) unknown op * in emobj ***(Assembler)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(813) unknown op * in size_psect ***(Assembler)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(815) syntax error in chipinfo file at line * ***(Assembler)***

The chipinfo file contains non-standard syntax at the specified line.

(817) unknown architecture in chipinfo file at line * ***(Assembler, Driver)***

An chip architecture (family) that is unknown was encountered when reading the chip INI file.

(829) unrecognized line in chipinfo file at line * ***(Assembler)***

The chipinfo file contains a processor section with an unrecognised line. Contact HI-TECH Support if the INI has not been edited.

(832) empty chip info file * ***(Assembler)***

The chipinfo file contains no data. If you have not manually edited the chip info file, contact HI-TECH Support with details.

(834) page width must be >= 60 ***(Assembler)***

The listing page width must be at least 60 characters. Any less will not allow a properly formatted listing to be produced, e.g.:

```
LIST C=10  ; the page width will need to be wider than this
```

(835) form length must be >= 15 ***(Assembler)***

The form length specified using the `-F`*`length`* option must be at least 15 lines. Setting this length to zero is allowed and turns off paging altogether. The default value is zero (pageless).

(836) no file arguments ***(Assembler)***

The assembler has been invoked without any file arguments. It cannot assemble anything.

(838) refc == 0 in decref ***(Assembler, Optimiser)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(839) relocation too complex ***(Assembler)***

The complex relocation in this expression is too big to be inserted into the object file.

(840) phase error ***(Assembler)***

The assembler has calculated a different value for a symbol on two different passes. This is probably due to bizarre use of macros or conditional assembly.

(842) bad bit number ***(Assembler, Optimiser)***

A bit number must be an absolute expression in the range 0-7.

(844) lexical error ***(Assembler, Optimiser)***

An unrecognized character or token has been seen in the input.

(845) multiply defined symbol * *(Assembler)*

This symbol has been defined in more than one place. The assembler will issue this error if a symbol is defined more than once in the same module, e.g.:

```
_next:
  move r0, #55
  move [r1], r0
_next:           ; woops -- choose a different name
```

The linker will issue this warning if the symbol (C or assembler) was defined multiple times in different modules. The names of the modules are given in the error message. Note that C identifiers often have an *underscore* prepended to their name after compilation.

(846) relocation error *(Assembler, Optimiser)*

It is not possible to add together two relocatable quantities. A constant may be added to a relocatable value, and two relocatable addresses in the same psect may be subtracted. An absolute value must be used in various places where the assembler must know a value at assembly time.

(847) operand error *(Assembler, Optimiser)*

The operand to this opcode is invalid. Check your assembler reference manual for the proper form of operands for this instruction.

(854) DS argument must be a positive constant *(Assembler)*

The argument to the `DS` assembler directive must be a positive constant, e.g.:

```
DS  -3  ; you cannot reserve a negative number of bytes
```

(857) psect may not be local and global *(Linker)*

A local psect may not have the same name as a global psect, e.g.:

```
psect text,class=CODE       ; text is implicitly global
  move r0, r1
; elsewhere:
psect text,local,class=CODE
  move r2, r4
```

The `global` flag is the default for a psect if its scope is not explicitly stated.

(862) symbol is not external *(Assembler)*

A symbol has been declared as EXTRN but is also defined in the current module.

(864) SIZE= must specify a positive constant *(Assembler)*

The parameter to the `PSECT` assembler directive's `size` option must be a positive constant number, e.g.:

```
PSECT text,class=CODE,size=-200  ; a negative size?
```

(865) psect size redefined *(Assembler)*

The `size` flag to the `PSECT` assembler directive is different from a previous `PSECT` directive, e.g.:

```
psect spdata,class=RAM,size=400
; elsewhere:
psect spdata,class=RAM,size=500
```

(867) psect reloc redefined *(Assembler)*

The `reloc` flag to the `PSECT` assembler directive is different from a previous `PSECT` directive, e.g.:

```
psect spdata,class=RAM,reloc=4
; elsewhere:
psect spdata,class=RAM,reloc=8
```

(868) DELTA= must specify a positive constant *(Assembler)*

The parameter to the `PSECT` assembler directive's `DELTA` option must be a positive constant number, e.g.:

```
PSECT text,class=CODE,delta=-2  ; a negative delta value does not make sense
```

(871) SPACE= must specify a positive constant *(Assembler)*

The parameter to the PSECT assembler directive's `space` option must be a positive constant number, e.g.:

```
PSECT text,class=CODE,space=-1  ; space values start at zero
```

(872) psect space redefined ***(Assembler)***

The `space` flag to the `PSECT` assembler directive is different from a previous `PSECT` directive, e.g.:

```
psect spdata,class=RAM,space=0
; elsewhere:
psect spdata,class=RAM,space=1
```

(875) bad character constant in expression ***(Assembler,Optimizer)***

The character constant was expected to consist of only one character, but was found to be greater than one character or none at all. An assembler specific example:

```
mov  r0, #'12'    ; '12' specifies two characters
```

(876) syntax error ***(Assembler, Optimiser)***

A syntax error has been detected. This could be caused a number of things.

(906) bad * memory option specification ***(Driver)***

The arguments to the memory option (e.g. `--RAM`) were badly formed, e.g.:

```
--RAM=0-
```

The high address is missing. Maybe you meant:

```
--RAM=0-1fffh
```

(915) no room for arguments ***(Preprocessor, Parser, Code Generator, Linker, Objtohex)***

The code generator could not allocate any more memory.

(916) can't allocate memory for arguments ***(Preprocessor, Parser, Code generator, Assembler)***

The compiler could not allocate any more memory when trying to read in command-line arguments.

(917) argument too long ***(Preprocessor, Parser)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(918) *: no match ***(Preprocessor, Parser)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(921) can't open chipinfo file * ***(Driver, Assembler)***

The chipinfo file could not be opened. This file normally resides in the LIB directory of the compiler distribution. If driving the assembler directly (without the command line driver) ensure that the option to location this file correctly specifies the path, otherwise contact HI-TECH Support with details.

(941) bad * assignment; USAGE: * ***(Hexmate)***

An option to Hexmate was incorrectly used or incomplete. Follow the usage supplied by the message and ensure that that the option has been formed correctly and completely.

(942) unexpected character on line * of file * ***(Hexmate)***

File contains a character that was not valid for this type of file, the file may be corrupt. For example, an Intel hex file is expected to contain only ASCII representations of hexadecimal digits, colons (:) and line formatting. The presence of any other characters will result in this error.

(944) data conflict at address *h between * and * ***(Hexmate)***

Sources to Hexmate request differing data to be stored to the same address. To force one data source to override the other, use the '+' specifier. If the two named sources of conflict are the same source, then the source may contain an error.

(945) checksum range (*h to *h) contained an indeterminate value ***(Hexmate)***

The range for this checksum calculation contained a value that could not be resolved. This can happen if the checksum result was to be stored within the address range of the checksum calculation.

(948) checksum result width must be between 1 and 4 bytes ***(Hexmate)***

The requested checksum byte size is illegal. Checksum results must be within 1 to 4 bytes wide. Check the parameters to the -CKSUM option.

(949) start of checksum range must be less than end of range ***(Hexmate)***

The -CKSUM option has been given a range where the start is greater than the end. The parameters may be incomplete or entered in the wrong order.

(951) start of fill range must be less than end of range ***(Hexmate)***

The -FILL option has been given a range where the start is greater than the end. The parameters may be incomplete or entered in the wrong order.

(953) unknown -HELP sub-option: * ***(Hexmate)***

Invalid sub-option passed to -HELP. Check the spelling of the sub-option or use -HELP with no sub-option to list all options.

(954) incomplete -O option; no file specified ***(Hexmate)***

The output filename option did not contain a filename. A filename must follow -O. Make sure the filename and -O are not separated by a space.

(956) -SERIAL value must be between 1 and * bytes long ***(Hexmate)***

The serial number being stored was out of range. Ensure that the serial number can be stored in the number of bytes permissible by this option.

(958) too many input files specified; * file maximum ***(Hexmate)***

Too many file arguments have been used. Try merging these files in several stages rather than in one command.

(960) unexpected record type(*) on line * of "*" ***(Hexmate)***

Intel hex file contained an invalid record type. Consult the Intel hex format specification for valid record types.

(962) forced data conflict at address *h between * and * ***(Hexmate)***

Sources to Hexmate force differing data to be stored to the same address. More than one source using the '+' specifier store data at the same address. The actual data stored there may not be what you expect.

(963) checksum range includes voids or unspecified memory locations ***(Hexmate)***

Checksum range had gaps in data content. The runtime calculated checksum is likely to differ from the compile-time checksum due to gaps/unused byes within the address range that the checksum is calculated over. Filling unused locations with a known value will correct this.

(966) no END record for HEX file "*" ***(Hexmate)***

Intel hex file did not contain a record of type END. The hex file may be incomplete.

(967) unused function definition: * (from line *) ***(Parser)***

The indicated `static` function was never called in the module being compiled. Being static, the function cannot be called from other modules so this warning imples the function is never used. Either the function is redundant, or the code that was meant to call it was excluded from compilation or misspelt the name of the function.

(968) unterminated string ***(Assembler, Optimiser)***

A string constant appears not to have a closing quote missing.

(969) end of string in format specifier ***(Parser)***

The format specifier for the printf() style function is malformed.

(970) character not valid at this point in format specifier ***(Parser)***

The printf() style format specifier has an illegal character.

(971) type modifiers not valid with this format ***(Parser)***

Type modifiers may not be used with this format.

(972) only modifiers h and l valid with this format ***(Parser)***

Only modifiers `h` (`short`) and `l` (`long`) are legal with this `printf` format specifier.

(973) only modifier l valid with this format ***(Parser)***

The only modifier that is legal with this format is `l` (for long).

(974) type modifier already specified ***(Parser)***

This type modifier has already be specified in this type.

(975) invalid format specifier or type modifier ***(Parser)***

The format specifier or modifier in the printf-style string is illegal for this particular format.

(976) field width not valid at this point ***(Parser)***

A field width may not appear at this point in a printf() type format specifier.

(978) this is an enum ***(Parser)***

This identifier following a `struct` or `union` keyword is already the tag for an enumerated type, and thus should only follow the keyword `enum`, e.g.:

```
enum IN {ONE=1, TWO};
struct IN {            /* woops -- IN is already defined */
  int a, b;
};
```

(979) this is a struct ***(Parser)***

This identifier following a `union` or `enum` keyword is already the tag for a structure, and thus should only follow the keyword `struct`, e.g.:

```
struct IN {
  int a, b;
};
enum IN {ONE=1, TWO};  /* woops -- IN is already defined */
```

(980) this is a union ***(Parser)***

This identifier following a `struct` or `enum` keyword is already the tag for a `union`, and thus should only follow the keyword `union`, e.g.:

```
union IN {
  int a, b;
};
enum IN {ONE=1, TWO};  /* woops -- IN is already defined */
```

(981) pointer required ***(Parser)***

A pointer is required here, e.g.:

```
struct DATA data;
data->a = 9;      /* data is a structure, not a pointer to a structure */
```

(982) nxtuse(): unknown op: * ***(Optimiser,Assembler)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(984) type redeclared ***(Parser)***

The type of this function or object has been redeclared. This can occur because of two incompatible declarations, or because an implicit declaration is followed by an incompatible declaration, e.g.:

```
int a;
char a;  /* woops -- what is the correct type? */
```

(985) qualifiers redeclared ***(Parser)***

This function has different qualifiers in different declarations.

(988) number of arguments redeclared ***(Parser)***

The number of arguments in this function declaration does not agree with a previous declaration of the same function.

(989) module has code below file base of *h ***(Linker)***

This module has code below the address given, but the `-C` option has been used to specify that a binary output file is to be created that is mapped to this address. This would mean code from this module would have to be placed before the beginning of the file! Check for missing psect directives in assembler files.

(990) modulus by zero in #if, zero result assumed ***(Preprocessor)***

A modulus operation in a `#if` expression has a zero divisor. The result has been assumed to be zero, e.g.:

```
#define ZERO 0
#if FOO%ZERO    /* this will have an assumed result of 0 */
  #define INTERESTING
#endif
```

(991) integer expression required *(Parser)*

In an `enum` declaration, values may be assigned to the members, but the expression must evaluate to a constant of type `int`, e.g.:

```
enum { one = 1, two, about_three = 3.12 };  /* no non-int values allowed */
```

(992) can't find op *(Assembler, Optimiser)*

This is an internal compiler error. Contact HI-TECH Software technical support with details.

(1198) too many "*" specifications; * maximum *(Hexmate)*

This option has been specified too many times. If possible, try performing these operations over several command lines.

(1201) all FIND/REPLACE code specifications must be of equal width *(Hexmate)*

All find, replace and mask attributes in this option must be of the same byte width. Check the parameters supplied to this option. For example finding 1234h (2 bytes) masked with FFh (1 byte) will result in an error, but masking with 00FFh (2 bytes) will be Ok.

(1202) unknown format requested in -FORMAT: * *(Hexmate)*

An unknown or unsupported INHX format has been requested. Refer to documentation for supported INHX formats.

(1203) unpaired nibble in * value will be truncated *(Hexmate)*

Data to this option was not entered as whole bytes. Perhaps the data was incomplete or a leading zero was omitted. For example the value Fh contains only four bits of significant data and is not a whole byte. The value 0Fh contains eight bits of significant data and is a whole byte.

(1204) * value must be between 1 and * bytes long *(Hexmate)*

An illegal legth of data was given to this option. The value provided to this option exceeds the maximum or minimum bounds required by this option.

(1212) Found * (*h) at address *h *(Hexmate)*

The code sequence specified in a -FIND option has been found at this address.

can't create cross reference file * *(Assembler)*

The assembler attempted to create a cross reference file, but it could not be created. Check that the file's pathname is correct.

couldn't create error file: * *(Driver)*

The error file specified after the `-Efile` or `-E+file` options could not be opened. Check to ensure that the file or directory is valid and that has read only access.

duplicate arch for * in chipinfo file at line * *(Assembler, Driver)*

The chipinfo file has a processor section with multiple ARCH values. Only one ARCH value is allowed. If you have not manually edited the chip info file, contact HI-TECH Support with details.

duplicate lib for * in chipinfo file at line * *(Assembler)*

The chipinfo file has a processor section with multiple `LIB` values. Only one `LIB` value is allowed. If you have not manually edited the chip info file, contact HI-TECH Support with details.

duplicate romsize for * in chipinfo file at line * *(Assembler)*

The chipinfo file has a processor section with multiple ROMSIZE values. Only one ROMSIZE value is allowed. If you have not manually edited the chip info file, contact HI-TECH Support with details.

duplicate sparebit for * in chipinfo file at line * *(Assembler)*

The chipinfo file has a processor section with multiple SPAREBIT values. Only one SPAREBIT value is allowed. If you have not manually edited the chip info file, contact HI-TECH Support with details.

duplicate * for * in chipinfo file at line * *(Assembler, Driver)*

The chipinfo file has a processor section with multiple values for a field. Only one value is allowed per chip. If you have not manually edited the chip info file, contact HI-TECH Support with details.

duplicate zeroreg for * in chipinfo file at line * *(Assembler)*

The chipinfo file has a processor section with multiple ZEROREG values. Only one ZEROREG value is allowed. If you have not manually edited the chip info file, contact HI-TECH Support with details.

no arg to -o *(Assembler)*

The assembler requires that an output file name argument be supplied after the `-O` option. No space should be left between the `-O` and the filename.

psect * not loaded on 0x* boundary *(Linker)*

This psect has a relocatability requirement that is not met by the load address given in a `-p` option. For example if a psect must be on a 4K byte boundary, you could not start it at 100H.

absolute expression required *(Assembler)*

An absolute expression is required as an argument to the `IF` assembler directive.

bad -A option: * *(Driver)*

The format of a `-A` option to shift the ROM image was not correct. The `-A` should be immediately followed by a valid hex number, e.g.:

```
-A
```

What is the offset? Maybe you meant:

```
-A200See Section 13.7.2 for more details regarding this option.
```

bad bit address *(Assembler, Optimiser)*

The address supplied is not a bit-addressable portion of the XA. Bit addressable portions include the registers R0 to R15, direct RAM from 20h to 3Fh, and the on-chip SFRs from 400h to 43Fh.

bad bit expression ***(Optimiser)***

There is a bad bit expression in the assembler file.

bad fixup value ***(Optimiser)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

bad operand ***(Optimiser)***

This operand is invalid. Check the syntax.

bad operand to SEG ***(Assembler, Optimiser)***

This can happen if you try to take the segment part of something that is already a segment address.

bit number not absolute ***(Optimiser)***

A bit number must be an absolute number in the range 0-7.

bit range check failed * ***(Linker)***

The assembler can place checks associated with an instruction in the output object file that will confirm that the value ultimately assigned to a symbol used within the instruction is within some range. This error indicates that the range check failed, i.e. the value was either too large or too small. This error relates to checks carried on a bit addresses. If there is no hand-written assembler code in this program, then this may be an internal compiler error and you should contact HI-TECH support with details of the code that generated this error. Other causes are numerous.

can't have arrays of bits ***(Code Generator)***

You can't have an array of bits, because bits can't be indexed.

can't have pointer to bit ***(Code Generator)***

Bit variables as implemented in the 8051 compiler are not addressable via pointers, so a pointer to a bit is not allowed.

can’t open include file * ***(Assembler)***

The named assembler include file could not be opened. Confirm the spelling and path of the file specified in the INCLUDE directive, e.g.:

```
INCLUDE "misspilt.h"  ; is the filename correct?
```

chip name * not found in chipinfo file ***(Driver)***

The chip type specified on the command line was not found in the chipinfo INI file. The compiler doesn’t know how to compile for this chip. If this is a device not yet supported by the compiler, you might be able to add the memory specifications to the chipinfo file and try again.

def[bmsf] in text psect ***(Optimiser)***

The assembler file supplied to the optimizer is invalid.

delete what ? ***(Libr)***

The librarian requires one or more modules to be listed for deletion when using the `d` key, e.g.:

```
libr d c:\ht-pic\lib\pic704-c.lib
```

does not indicate which modules to delete. try something like:

```
libr d c:\ht-pic\lib\pic704-c.lib wdiv.obj
```

direct range check failed * ***(Linker)***

The assembler can place checks associated with an instruction in the output object file that will confirm that the value ultimately assigned to a symbol used within the instruction is within some range. This error indicates that the range check failed, i.e. the value was either too large or too small. If there is no hand-written assembler code in this program, then this may be an internal compiler error and you should contact HI-TECH support with details of the code that generated this error. Other causes are numerous.

duplicate banks for * in chipinfo file at line * ***(Assembler)***

The chipinfo file has a processor section with multiple BANKS values. Only one BANKS value is allowed. If you have not manually edited the chip info file, contact HI-TECH Support with details.

end statement inside include file or macro ***(Assembler)***

An `END` statement was found inside an include file or a macro.

expression error ***(Assembler, Optimiser)***

There is a syntax error in this expression.

flag * unknown ***(Assembler)***

This option used on a `PSECT` directive is unknown to the assembler.

floating number expected ***(Assembler)***

The arguments to the `DEFF` pseudo-op must be valid floating point numbers.

function's local data too large ***(Code Generator)***

The size of the stack frame for this function is greater than that allowable. The size is limited by the size of the internal RAM on the 8051.

garbage after operands ***(Assembler)***

There is something on this line after the operands other than a comment. This could indicate an operand error.

garbage on end of line ***(Assembler)***

There were non-blank and non-comment characters after the end of the operands for this instruction. Note that a comment must be started with a semicolon.

identifier expected ***(Parser)***

Inside the braces of an `enum` declaration should be a comma-separated list of identifiers, e.g.:

```
enum { 1, 2};  /* woops -- maybe you mean enum { one = 1, two }; */
```

incomplete ident record ***(Libr)***

The IDENT record in the object file was incomplete. Contact HI-TECH Support with details.

incomplete symbol record ***(Libr)***

The SYM record in the object file was incomplete. Contact HI-TECH Support with details.

invalid bit address: * ? ***(Code Generator)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

invalid qualifier combination on * ***(Code Generator)***

This qualifier combination is illegal, perhaps because it is contradictory.

label not followed by : ***(Optimiser)***

The optimizer has encountered a syntax error in its input.

library file names should have .lib extension: * ***(Libr)***

Use the `.lib` extension when specifying a library filename.

line too long ***(Optimiser)***

This line is too long. It will not fit into the compiler's internal buffers. It would require a line over 1000 characters long to do this, so it would normally only occur as a result of macro expansion.

module * defines no symbols ***(Libr)***

No symbols were found in the module's object file. This may be what was intended, or it may mean that part of the code was inadvertently removed or commented.

no RAM areas defined ***(Driver)***

The `-RAM` options was invoked but no valid bank address ranges were present.

no ROM banks defined ***(Driver)***

The `-ROM` options was invoked but no valid bank address ranges were present.

oops! -ve number of nops required! ***(Assembler)***

This is an internal compiler error. Contact HI-TECH Software technical support with details.

phase error in macro args ***(Assembler)***

The assembler has detected a difference in the definition of a symbol on the first and a subsequent pass.

psect limit redefined ***(Assembler)***

The psect limit has already been defined using the psect `limit` flag elsewhere, e.g.:

```
psect text,class=CODE,limit=1ffh
  move r0, r1
; elsewhere:
psect text,class=CODE,limit=2ffh
  move r2, r4
```

RAM area * low bound greater than high bound ***(Driver)***

An additional memory bank has been defined which has a lower address bound greater than the high address bound.

RAM area * out of range for chip * ***(Driver)***

An additional memory bank has been defined which does not fall into the available valid ranges for this chip.

replace what ? ***(Libr)***

The librarian requires one or more modules to be listed for replacement when using the `r` key, e.g.:

```
libr r lcd.lib
```

This command needs the name of a module (`.obj` file) after the library name.

reserved * area and reserved ICD * range overlap in region * ***(Driver)***

The -ICD option has been used which reserves memory locations for the debugger. Additional memory areas have been reserved with the `-RESROM` or `-RESRAM` option and these address ranges overlap those required by the ICD.

restore without matching save ***(Assembler)***

The RESTORE assembler control directive has been used without a preceding SAVE assembler control directive.

save/restore too deep ***(Assembler)***

Too many `SAVE` assembler control directives have been used.

symbol has been declared extern ***(Assembler)***

A symbol has been declared in the current module, but has previously been declared extern. A symbol cannot be both local and extern, e.g.:

```
extern _foo   ; this shouldn't be specified if the symbol is defined here
foo:
  goto start
```

too many object files ***(Driver)***

A maximum of 128 object files may be passed to the linker. The driver exceeded this amount when generating the command line for the linker.

too many operands ***(Optimiser)***

There are too many operands to this instruction.

too many symbols in * ***(Optimiser)***

There are too many symbols in the specified function. Reduce the size of the function.

undefined public symbol * ***(Assembler)***

A symbol has been declared `PUBLIC` but has not been defined.anywhere in the module.

unknown directive ***(Assembler)***

An unknown assembler control directive was used.

unknown psect ***(Optimiser)***

The assembler file read by the optimizer has an unknown psect.

too many common lines in chipinfo file for * ***(Assembler, Driver)***

The chipinfo file contains a processor section with too many `COMMON` fields. Only one `COMMON` field is allowed per processor.

Appendix C

Chip information

Couldn’t open input file: ../../lib/8051-c.ini

Appendix D

Regular Expressions

Expression	Matches
Characters	
x	The character *x*
\\	The backslash character
\0*n*	The character with octal value 0*n* (0 <= *n* <= 7)
\0*nn*	The character with octal value 0*nn* (0 <= *n* <= 7)
\0*mnn*	The character with octal value 0*mnn* (0 <= *m* <= 3, 0 <= *n* <= 7)
\x*hh*	The character with hexadecimal value 0x*hh*
\u*hhhh*	The character with hexadecimal value 0x*hhhh*
\t	The tab character ('\x09')
\n	The newline (line feed) character ('\x0A')
\r	The carriage-return character ('\x0D')
\f	The form-feed character ('\x0C')
\a	The alert (bell) character ('\x07')
\e	The escape character ('\x1B')
\c*x*	The control character corresponding to *x*
Character classes	
[abc]	a, b, or c (simple class)
[^abc]	Any character except a, b, or c (negation)
[a-zA-Z]	a through z or A through Z, inclusive (range)
[a-d[m-p]]	a through d, or m through p: [a-dm-p] (union)
continued…	

Expression	Matches
[a-z&&[def]]	d, e, or f (intersection)
[a-z&&[^bc]]	a through z, except for b and c: [ad-z] (subtraction)
[a-z&&[^m-p]]	a through z, and not m through p: [a-lq-z](subtraction)
Predefined character classes	
.	Any character (may or may not match line terminators)
\d	A digit: [0-9]
\D	A non-digit: [^0-9]
\s	A whitespace character: [\t\n\x0B\f\r]
\S	A non-whitespace character: [^\s]
\w	A word character: [a-zA-Z_0-9]
\W	A non-word character: [^\w]
POSIX character classes (US-ASCII only)	
\p{Lower}	A lower-case alphabetic character: [a-z]
\p{Upper}	An upper-case alphabetic character:[A-Z]
\p{ASCII}	All ASCII:[\x00-\x7F]
\p{Alpha}	An alphabetic character:[\p{Lower}\p{Upper}]
\p{Digit}	A decimal digit: [0-9]
\p{Alnum}	An alphanumeric character:[\p{Alpha}\p{Digit}]
\p{Punct}	Punctuation: One of !"#$%&’()*+,-./:;<=>?@[\]^_‘{|}~
\p{Graph}	A visible character: [\p{Alnum}\p{Punct}]
\p{Print}	A printable character: [\p{Graph}]
\p{Blank}	A space or a tab: [\t]
\p{Cntrl}	A control character: [\x00-\x1F\x7F]
\p{XDigit}	A hexadecimal digit: [0-9a-fA-F]
\p{Space}	A whitespace character: [\t\n\x0B\f\r]
Classes for Unicode blocks and categories	
\p{InGreek}	A character in the Greek block (simple block)
\p{Lu}	An uppercase letter (simple category)
\p{Sc}	A currency symbol
\P{InGreek}	Any character except one in the Greek block (negation)
[\p{L}&&[^\p{Lu}]]	Any letter except an uppercase letter (subtraction)
Boundary matchers	
^	The beginning of a line
$	The end of a line
\b	A word boundary
continued…	

Expression	Matches
\B	A non-word boundary
\A	The beginning of the input
\G	The end of the previous match
\Z	The end of the input but for the final terminator, if any
\z	The end of the input
Greedy quantifiers	
X?	*X*, once or not at all
*X**	*X*, zero or more times
X+	*X*, one or more times
X{*n*}	*X*, exactly *n* times
X(*n*,}	*X*, at least *n* times
X{*n*,*m*}	*X*, at least *n* but not more than *m* times
Reluctant quantifiers	
X??	*X*, once or not at all
*X**?	*X*, zero or more times
X+?	*X*, one or more times
X{*n*}?	*X*, exactly *n* times
X(*n*,}?	*X*, at least *n* times
X{*n*,*m*}?	*X*, at least *n* but not more than *m* times
Possessive quantifiers	
X?+	*X*, once or not at all
*X**+	*X*, zero or more times
X++	*X*, one or more times
X{*n*}+	*X*, exactly *n* times
X(*n*,}+	*X*, at least *n* times
X{*n*,*m*}+	*X*, at least *n* but not more than *m* times
Logical operators	
XY	*X* followed by *Y*
X|*Y*	Either *X* or *Y*
(*X*)	*X*, as a capturing group
Back references	
n	Whatever the *n*th capturing group matched
Quotation	
\\	Nothing, but quotes the following character
\Q	Nothing, but quotes all characters until \E
continued…	

Expression	Matches
\E	Nothing, but ends quoting started by \Q
Special constructs (non-capturing)	
(?:*X*)	*X*, as a non-capturing group
(?idmsux-idmsux)	Nothing, but turns match flags on - off
(?idmsux-idmsux:X)	X, as a non-capturing group with the given flags on - off
(?=*X*)	*X*, via zero-width positive lookahead
(?!*X*)	*X*, via zero-width negative lookahead
(?<=*X*)	*X*, via zero-width positive lookbehind
(?<!*X*)	*X*, via zero-width negative lookbehind
(?>*X*)	*X*, as an independent, non-capturing group

Index